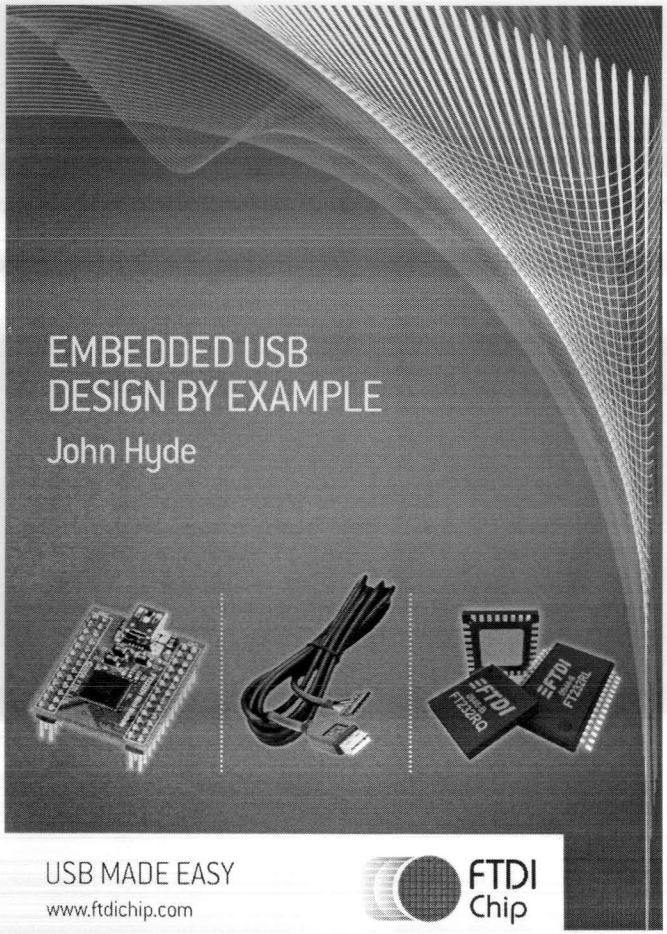


Embedded USB Design By Example

John Hyde

(Commissioned by FTDI Ltd)

Embedded USB Design By Example

John Hyde

Foreword by Fred Dart – Founder and CEO of FTDI.

John Hyde is an internationally recognised and renowned figure in the field of USB, having authored the seminal "USB Design By Example" series of books which have helped many engineers understand the underlying complexity of USB by leading them through a series of practical examples.

I am delighted that John has undertaken to author a new book, Embedded USB Design By Example, at our behest for those of us who would like to incorporate USB interfacing into their product designs whilst focussing on overall product development concepts rather than having to learn the intricacies of USB hardware and driver development. Written in John's unique style, this book is intended as a supplement to the existing data sheets and application notes on our FTDI web site.

Future Technology Devices International Limited, aka FTDI, is a well known semiconductor supplier in the USB "legacy" device field. Our FT232, FT245 and Hi speed dual and quad device series of USB peripheral devices offer a seamless route for easy USB interfacing through proven, well understood serial and parallel interfaces. Coupled with a commitment to providing royalty free, multi-platform USB drivers developed in house to ensure quality and consistency, our USB interface solutions can dramatically improve time to market for USB product designs eliminating ongoing support costs in driver development.

FTDI's Vinculum Host/Peripheral controller range offers the same approach for embedded products that require USB Host capability. These will be covered in Part 2 of the book which will be launched later in 2010.

For further details of FTDI's USB solutions, please visit our website www.ftdichip.com .

Acknowledgements:

I was warmly welcomed to FTDI's head office in Glasgow, UK, where a large group of enthusiastic engineers provided me more information than I could ever have asked for. I would like to particularly thank Fred and Cathy Dart for their hospitality and Ian Dunn for organizing the presentations, training, interviews, and material reviews. While every-ones contributions are valued I alone am responsible for any errors - I welcome your feedback (to John@usb-by-example.com) so that this book may be improved in future revisions.

I must thank my family, Lorraine, BJ and CJ who all contributed to the creation of this book – I appreciate all of their hard work, support and encouragement.

I have been involved with USB since its invention and I applaud FTDI's efforts to make USB easy - I trust that with the help of this project-orientated book and FTDI's components that even more people will be able to benefit from the worlds most popular bus, USB.

Table of Contents:

Part 1

Chapter 1 Introduction and Essential USB theory
- USB History
- USB Architecture
- Importance of a USB hub

Chapter 2 – A starter USB project
- How It Works
- Getting More IO lines

Chapter 3 - Serial and parallel device conversion
- Representative Serial Device
- Windows Conversion
- Mac OS X Conversion
- Converting a Parallel Device

Chapter 4 - Connecting to more capable devices.
- Data Collection Pod
- Dual USB-to-SPI Adaptor
- USB-to-Custom Parallel Adaptor

Part 2

Chapter 5: Vinculum-I Design Examples
- Adding a Flash Drive to a product
- JPEG viewer and MPEG player
- Portable data logger
- Embedded flash drive designs now enabled

Chapter 6: Getting to know Vinculum-II

Chapter 7: Writing software for the Vinculum-II
- Multitasking RTOS 101
- Buttons and Lights using VOS
- Vinculum-II Device Driver Architecture

Chapter 8: Using the Vinculum-II IDE (available late May)

Chapter 9: Building a 'Smart Device' (available late May)

Chapter 10: Interconnecting two USB devices (available late May)

Please register your book at the following e-mail address so that updates may be sent when available:

Designbyexamplepart2@ftdichip.com

List of Figures:

Part 1

Figure 1.1: USB structure from USB specification[Ref 1]
Figure 1.2: USB is a 4-wire, serial, point-to-point connection
Figure 1.3: Descriptors are fixed-format blocks of data
Figure 1.4: A USB hub provides connectivity
Figure 1.5: Descriptors for a basic hub
Figure 1.6: A High-Speed hub includes Transaction Translators
Figure 1.7: Typical PC with several hubs
Figure 2.1: The TTL-232R is a USB-to-4BitIO port cable
Figure 2.2: The first example, schematic and hardware
Figure 2.3: Edited source code of first example
Figure 2.4: Block diagram of FT232R USB-ByteMover device
Figure 2.5A: Showing detail of ABUS data routing
Figure 2.5B: Showing detail of programmable IO pins
Figure 2.6: Adding an I2C IO expander to the cable
Figure 2.7: Waveform needed to read an I2C byte
Figure 2.8: 2-way I2C bus expansion using PCA9554
Figure 2.9: 4-way I2C bus expansion using MCP23008
Figure 3.1: Representative Serial Device
Figure 3.2: Connecting the FTDI cable to the display
Figure 3.3: The cable is recognized by Windows as a COM port
Figure 3.4: The cable is recognized by Mac OS X as a COM port
Figure 3.5: Converting a serial device
Figure 4.1: Block diagram of FT2232H
Figure 4.2: Block diagram of data collection pod
Figure 4.3: Options for adding USB
Figure 4.4: Block diagram of 'Reader' plus 'Pods'
Figure 4.5: Detail of Channel A IO connections
Figure 4.6: FT-2232H mini module used for prototyping
Figure 4.7: MPSSE commands used to drive SPI
Figure 4.8: USBee trace of GetDeviceID SPI command
Figure 4.9: Control signals used by most LCD character displays
Figure 4.10: Single channel DataPod using a DLP-1232H module

Part 2:

Figure 5.1: Vinculum-I operates as an attached device
Figure 5.2: Vinculum-I uses a DOS-like command interface
Figure 5.3: USB-to-Serial cable connected to VMusic board
Figure 5.4: Some of the monitor's DOS-like commands
Figure 5.5: This example was developed and debugged using a PSoC development system
Figure 5.6: The DLP-VLOG showcases Vinculum-I's capabilities
Figure 6.1: Vinculum-II supports standalone operation
Figure 6.2: Vinculum-II hardware block diagram
Figure 6.3: A debug module connects to your target system
Figure 6.4: The IO Mux connects peripherals to physical pins
Figure 6.5: Each IO pin has a configurable driver/receiver
Figure 7.1: Applications programming environments
Figure 7.2: A program has several tasks that interact
Figure 7.3: Tasks continuously move though this state diagram
Figure 7.4: Vinculum-II Software Block Diagram
Figure 7.5: Software Initialization Steps

Chapter 1 Introduction and Essential USB theory

Our electronics industry uses the term "embedded" to describe a non-reprogrammable, or fixed function, piece of equipment or device. This book also uses this definition but with an added, more literal, meaning. Most dictionaries define embedded as "enclosed firmly in a surrounding mass" and this is the approach that I will be taking with "Embedded USB." Yes, the designs will have USB inside but this is not their main focus. Most USB books describe USB as a technical wonder (which it is) then flood the reader with an overwhelming amount of detail. I am not going to do that, so this book will NOT make you a USB technical guru. What it will do however, is describe USB as a tool that you can use to solve a wide array of problems.

I assume that you, the reader, have a basic understanding of electronics but that this is not your primary job function. You are tasked with building an industry-specific device that is not available off-the-shelf (else you would have purchased it and carried on with your real job!). This industry-specific device must interface to a PC (defined in this book as a personal computer running a USB-aware operating system such as Windows, OS X or Linux) and must therefore use an available PC IO connection. Or you have identified a very useful and cost effective PC peripheral, such as a joystick or flash drive, which you need to connect to your equipment. In both cases the connection standard is USB. THIS is what this book is about – how can you best utilize this USB connection to solve your specific problem. This book is example based and is divided into two parts – the first includes a wide range of example solutions that connect to a PC and the second part describes a wider range of example solutions that control USB-based PC peripheral devices.

I toyed with the idea of calling this book "USB for the rest of us" in deference to Apple's campaign around their introduction of the Macintosh computer. For those of you who don't remember the revolution Apple caused in 1984, they positioned the existing Wintel PC as difficult to use since you needed to know how it worked to be a successful user. Apple explained how you could be immediately productive with a Macintosh since its complexity was hidden behind an easy-to-use human interface that used a mouse and graphical display. My goal is similar – I want to show you that you can **use** USB **without knowing** its intricate details.

In this introductory chapter I review the facets of USB that you need to know to be successful. There have been several books and numerous papers written that describe the intricate details of USB, but, to be frank, you don't need to know most of this information to be able to use USB successfully. In the olden days, when USB was first introduced, you had to know these details since the available silicon components that you would use to implement a USB device were quite primitive, but today almost all of the complexity of USB has been integrated into fifth generation silicon devices that are straight forward to use. In fact, we will implement all of the examples in part 1 of this book without having to refer to the USB specification or other USB-specific documentation. We will use the skills you already have, such as interfacing with simpler serial buses (RS232, I2C, SPI, etc), and with parallel buses (FIFOs, 8051 MCU etc) to create a variety of USB-based solutions.

USB History

But first, a little history. It is important to know how we got here since this will enable us to move forward with more confidence. USB was invented in the mid 1990s to solve a specific problem – desktop PC peripheral device expansion. At this time the Wintel PC industry was stalled; Intel was producing microprocessors with ever-increasing performance but this could not be delivered to the peripheral devices; everyone wanted to use the Wintel PC as the computing engine to drive their custom peripheral device since this was cost-effective, but IO expansion in those days meant unique boards or connectors and custom device drivers. It was projected that there would not be enough software engineers available on the planet to support this expanding and diverging software need. Yes, "plug-and-play" had started to take hold but the existing PC infra structure of parallel ports, serial ports, EISA and PCI buses could not support emerging telephony and video-based applications. Something fundamentally different was required.

USB Architecture

The first USB design decision was to assign another microprocessor to handle the increasing IO load – this USB host controller would manage all of the low-level interactions of the peripheral devices thus freeing up the main CPU to process user applications data. USB would be a master-slave bus with a single master, the USB host controller, and multiple slaves, the IO devices. Most of the communications complexity would be implemented in the host controller, since there was only one, and this would allow the IO devices to be simpler and therefore lower cost. It was decided that

the USB host controller would have a 1ms scheduling period and that data transfers could be synchronized to this period – this would enable time-based data (audio and video for example) to be supported. The host CPU would generate lists of data transfers for each upcoming 1 ms time interval and the USB host controller would implement the data transfers on the host CPUs behalf. Once the host controller specification was agreed it was implemented as a fixed-function ASIC. This functional partitioning and standardization of IO functions prompted a new device driver model that enabled the low level USB data transfer mechanisms to be the same across a wide variety of peripheral devices – the diverging device driver problem had been contained!

As USB evolved so did the USB Host Controller specification. There are now three specifications (UHCI, OHCI, and EHCI) and there will soon be a fourth (XHCI). All are well defined with specifications downloadable from the web and all have been implemented in silicon. Each has proven and, in the case of Wintel PCs, WHQL certified OS device drivers. But the USB development team did not stop there – to ensure that the U in USB really meant ***Universal*** they divided the diversity of known and upcoming USB devices into CLASSes and then defined a set of standardized class interfaces above the standardized host controller interface. Microsoft, Apple, the Linux community and several silicon vendors then went about implementing a wide breadth of standardized drivers. The benefit to the IO device developers is enormous – if they implemented the interfaces on their devices to match the USB class specifications then they would operate immediately with all operating systems that implemented the class driver. These standardized implementations mean that a keyboard, modem, flash drive, printer, etc can be moved around different platforms and will continue to perform as designed. Also, since all communications is protocol based it will be simple to swap out the hardware device with something faster, cheaper, or more capable. Software did not have to be redone so the large investment in applications software could be preserved.

I have taught USB to many people and a great number get hung up on a key diagram from the USB specification – Figure 5.9 reproduced (with permission) as my Figure 1.1. It is essential that you understand this figure so let's study it for a moment since it unlocks much of the insight required to conquer USB and use it as a tool.

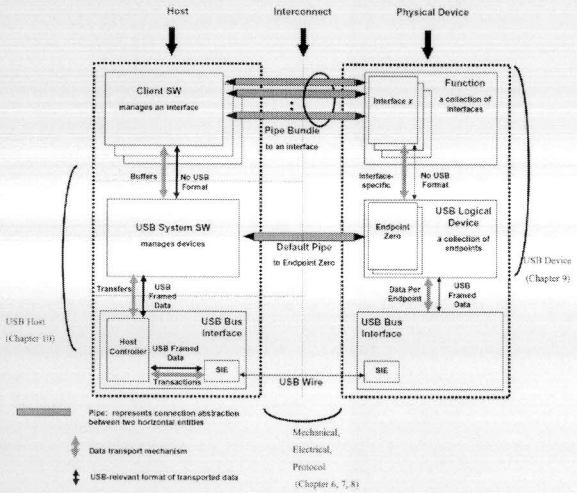


Figure 1.1: USB structure from USB specification[Ref 1]

Figure 1.1 shows two dotted boxes, ***Host*** and ***Physical Device***, interconnected with a ***USB wire***. It is important to realize that the Host (typically a PC) contains two or more CPUs and the Physical Device contains one or more CPUs – these CPUs can be reprogrammable (like the x86 Intel Processor in today's PC) or fixed function implemented as a ASIC (like the USB Host Controller) but note that they are smart. And we have a smart USB interconnecting this smart multi-CPU environment. It all looks a little daunting but, fear not, you do not need to know exactly how this all works to be successful using USB.

Figure 1.1 is a run-time diagram – it assumes that the connection between the Host and the Physical Device has already been set up (this is described later in this chapter). Data transfer starts with the Host so, as an example, let's send an ASCII string

"Hello World" from the Host to the Physical Device. We inject this 11-byte array into the ***Client SW*** block using a system call such as WriteFile( ). The Client SW may break our data into multiple buffers as it sends it to the ***USB System SW*** block. Note that this USB System SW block is receiving data transfer requests from many instantiations of Client SW blocks within different applications running on the PC. USB is a shared media but each application program treats it as a personal data connection; it is the USB System SW that manages the multiple data transfer requests from all of the Client SW blocks and it constructs a table of the ***Transfers*** necessary to service all of the requests. The USB System SW calculates the needed data transfers using a 1ms-scheduling period. The PC's x86 processor then passes this list of ***USB Framed Data*** to the USB Host Controller.

The USB Host Controller manages the low-level signaling on the ***USB wire***. It embeds our "Hello World" user data into one or more ***Transactions*** using asynchronous packets, which also includes SYNC data, device addressing data and error checking data. The ***Serial Interface Engine*** (SIE in Figure 1.1) handles automatic error retries which results in reliable data transfer between the Host and the Physical Device. The USB Interface on the Physical Device monitors all traffic on the USB wire and if it detects a packet with its assigned address then it absorbs and checks the packet and passes validated packets up to the ***USB Logical Device***. The USB Logical Device will pass user data packets up to the ***Function Block*** and our "Hello World" data will appear at the top of our Physical device.

Now focus on the horizontal bars called ***Pipe Bundle*** in Figure 1.1. The Host pushed the "Hello World" data into the top of the Host stack and it popped out of the top of the Physical Device stack. *It appeared to travel through the horizontal Pipe Bundle.* In reality it went all down the Host stack, across the USB wire and all the way up the Physical Device stack but we need not be concerned about this. The lowest level of Figure 1.1 (USB Interface, SIE and Host Controller) is fully defined by the USB specification and is implemented in fixed-function silicon. The center layer is also fully defined by the USB specification and is implemented in software, firmware or hardware (the Default Pipe is used for Link Management and is described later). The upper level is also fully defined by the USB specification and therefore, like the other layers, you have no flexibility to change it. It is interesting to know how this CPU-to-CPU communications is implemented but this knowledge is not necessary to use USB – if you accept that data effectively moves from a buffer in the Host system into a buffer in the Physical Device system (and visa

versa) then the key questions are; what is the latency, and what is the data throughput. We will address these questions in the examples chapters.

You could ask "but how do I differentiate my product within this standardized market place?" If you have the time and funds you can implement using "vendor defined" interfaces which are included within the USB specification as an option. But while you are moving along this difficult and time-consuming path don't be surprised if your competitor introduces a similar product using a collection of standard drivers and captures most of the available customer base. I am a STRONG advocate of OS-supplied and vendor-supplied drivers and always recommend that everyone use this route. I maintain that you need an extremely compelling reason to embark on writing your own device driver; most people don't.

The second major design decision made by the USB creators was the interconnection scheme. For ease of implementation and lowest cost, a 4-wire, serial, point-to-point connection, as shown in Figure 1.2 was chosen. A USB cable has an 'A' end (upstream connector, towards the host) and a 'B' end (downstream connector, towards the device). The 'A' connector included a +5V power source that could be used by a peripheral device and this could eliminate the need of many devices to include their own power (from a wall wart for example). There are rules to the amount of power that can be supplied and these are discussed later. The two signal wires are a half-duplex, differential pair that are generally driven by the host controller – the direction is switched when the host needs to read from a device.

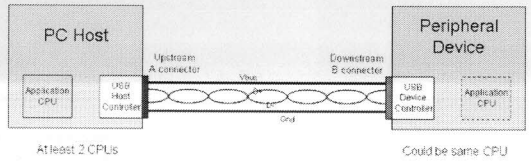


Figure 1.2: USB is a 4-wire, serial, point-to-point connection

Three standard signaling speeds are defined; low at 1.5Mb/sec, full at 12Mb/sec and high at 480Mb/s. There is a fourth option, SuperSpeed at ~5Gb/s, currently being developed by the USB Implementers Forum (USB IF). Information is transferred using asynchronous packets and these are combined with base protocols to implement four types of data transaction: control, interrupt, bulk, and isochronous. The USB specification also includes error checking and recovery mechanisms such that USB provides reliable data transfer – better still, this has been implemented in silicon by a variety of vendors so there is little reason to know every nuance. The USB IF has Compliance and Compatibility tests that silicon vendors must pass and this guarantees that the components we buy adhere to the USB specification.

Figure 1.2 shows a single USB Link connecting a USB host controller to a USB device. The host is required to support all three-link speeds (note that USB 1.1 compliant hosts will only support low and full speeds) and it is the device that selects the link speed. The USB specification includes link management commands that allow the host to interrogate the device to discover its identity and its capabilities. When the device is first connected, the host sends control transactions to the device to read pre-defined data blocks called ***Descriptors***, an example of which is shown in Figure 1.3.

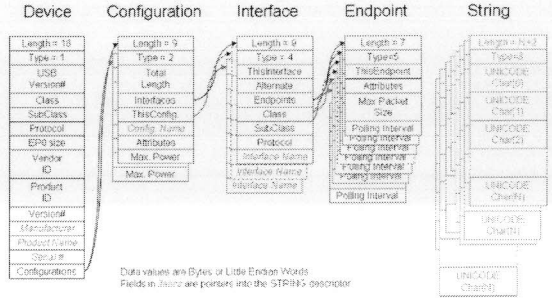


Figure 1.3: Descriptors are fixed-format blocks of data

The host operating system uses this descriptor information to determine which device driver should be used for each specific device and to assign a unique address to each attaching USB device. This process is called Enumeration and the requirement that each USB device is self-identifying is a major contributor to the plug-and-play and ease-of-use of USB. Most devices today implement the enumeration process in silicon or in canned firmware. So, once again, there is little need to understand every detail.

Importance of a USB hub

An integral part of the USB specification is a special device called a hub – this provides several bi-directional data repeaters and power injection as shown in Figure 1.4. This figure shows a USB 1.1 full/low speed hub since this is easier to explain (I cover a USB 2.0 high/full/low speed hub next). The hub contains a fixed-function USB device and the descriptors of a typical hub are shown in Figure 1.5. When this device is first attached to the host the operating system enumerates it and discovers that it is a hub – it therefore loads a hub device driver. This hub device driver manages the downstream connections. It applies power to each downstream port in turn and checks to see if a device is attached – an attached device will change the DC state of the data lines. If a device is detected then the hub device connects the downstream port to the upstream port and the host enumerates this new device – from this stage onwards the new device does not know that it is connected to the host via a hub, this is a transparent connection (yes, there is a small propagation delay through the hub but this is allowed for in the spec). The new device operates as if it were directly connected to the host. If the new device was another hub then the process would repeat – the USB specification allows for hubs up to five deep, which gives a lot of connectivity.

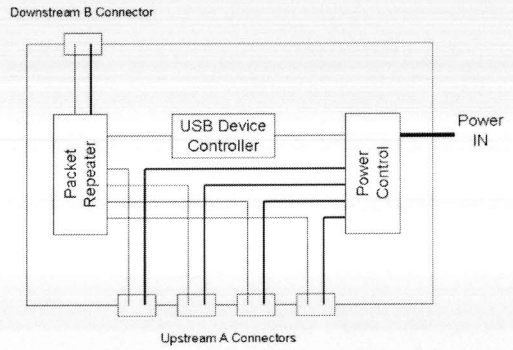


Figure 1.4: A USB hub provides connectivity

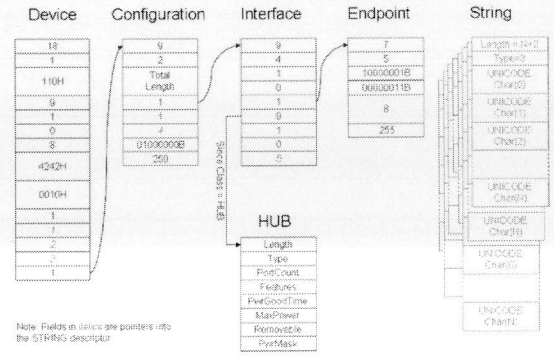


Figure 1.5: Descriptors for a basic hub

The hub also allows for the injection of power into downstream ports. The USB specification details several power levels; when first connected a device can draw up to 100mA from the upstream connection; during enumeration the device can request up to 500mA (if your device needs more then 500mA then it will need its own power source); when suspended, or not operating, a device must limit its power drain to less than 2.5mA – a PC may suspend itself and power down when not in use, and there is no point in having peripheral devices powered up when the PC is off. So the USB host controller will suspend all attached devices prior to powering down.

The basic functionality of a USB 2.0 hub, as shown in Figure 1.6, is the same as a USB 1.1 hub. Additional circuitry is included that enables more efficient use of the USB Links. A high-speed link always runs at high speed – if a low or full speed device is connected to a high-speed hubs downstream port then data transfers are "stored-and-forwarded." The data packets are sent at high speed from the host to a ***Transaction Translator*** (TT in Figure 1.6), which will then send the packet at low or full speed to the device. Similarly responses are collected at low or full speed by the TT and forwarded to the host at high speed. These operations require additional link management commands (Start Split etc.) and these are implemented by the EHCI hub driver at the host. There is no additional programming at the PC application layer nor at the device so these operations are transparent to the device and to the user.

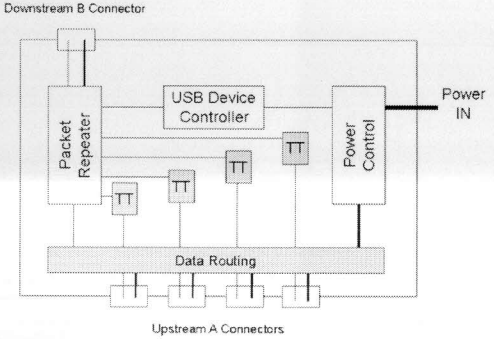


Figure 1.6: A High-Speed hub includes Transaction Translators

Figure 1.7 shows a typical PC system with a variety of devices attached via hubs. Each EHCI host controller will support a 480Mb/sec link and this bandwidth is shared by all of the devices connected via this link. Each OHCI/UHCI host controller will support a 12Mb/sec link and again this bandwidth is shared by downstream devices. A PC typically has multiple host controllers; the laptop I am using at the moment has an Intel ICH9 controller which includes 6 UHCI controllers and 2 EHCI controllers. The ICH9 also has on chip routing and the operating system will assign a UHCI controller to manage low/full speed devices and it reserves the EHCI controller connections for high speed devices. If the OS cannot route an EHCI controller to the port where a high speed device is connected it will prompt the user to move the device and plug it in elsewhere. Therefore this particular laptop can support up to 6*12+2*480=1Gb/sec of USB bandwidth.

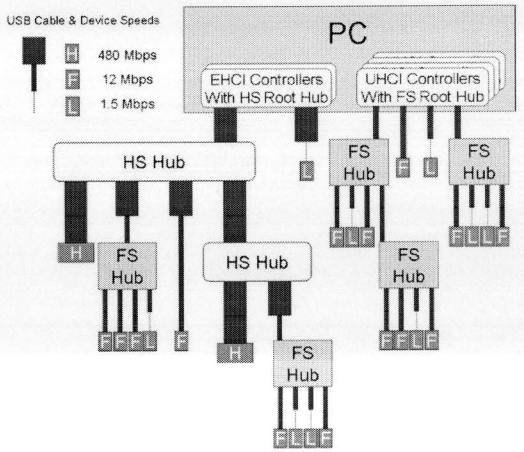


Figure 1.7: Typical PC with several hubs

Chapter Summary

In summary, USB is a shared communications media where multiple host controllers can be used to supply a desired IO bandwidth, and multiple hubs can be used to distribute this bandwidth to a diverse array of peripheral devices. All communications is standards-based and is implemented as a collection of proven class- and host controller drivers. The lower levels of this communication are implemented in fixed-function silicon. Since most of USB is standardized (and therefore cannot be changed) and most products are certified to be compliant to the USB specification then we can trust that USB works and focus our efforts on using USB to implement useful products.

So enough theory, lets implement something!!!

Part 1 focuses on designing IO devices that can be attached to a PC. I created a common source code for the Windows and Mac platforms, and I expect Linux users will be able to use the Mac OS X code with little or no modifications. I had to put an OS-specific #DEFINE to accommodate differences in library and some function names but, fundamentally, the **SAME** example code is running on all platforms. This is possible since FTDI provide their device drivers on all three platforms. I will focus on functionality and ease of understanding and not on the human interface so the code will be fundamental and written in portable C++. A set of PCBs is available (see Appendix B) to simplify working through the examples but if you don't have these then most of the examples can also be built up on a solder-less breadboard.

Ref 1: USB 2.0 Specification © 2000 Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, Philips. A free download is available from www.usb.org/developer.

Chapter 2 – A starter USB project

Let's start simple; you want to connect a single push button to a PC. On recognizing the button press, a program running on the PC initiates a series of actions one of which lights an LED adjacent to the push button (in case the PC is remote or does not have a screen, the LED provides feedback to the user that the button press has been recognized). This project could form the basis of an embedded kiosk, machine operation, sequence control, security monitoring or a range of other man-computer interactions. This project used to be easy when the PC had a parallel port but all you see now are USB ports! But you don't have time to learn USB, so you look online to buy a USB-to-ButtonAndLight adaptor that you can just use. Nobody sells one. HELP!!!

Fortunately FTDI sells and supports a USB-to-4BitIOport cable that can be used to solve this problem. FTDI don't call it that (they call it a TTL-232R) but that is how we shall use it and it is shown in Figure 2.1. It looks like a standard USB cable until you look closely at the non-USB end – there are six wires instead of the expected four. Two are power (+5V) and ground and the other four are TTL signals that can be configured as inputs or outputs. There is a fixed-function USB device molded into the plug but more of this later.

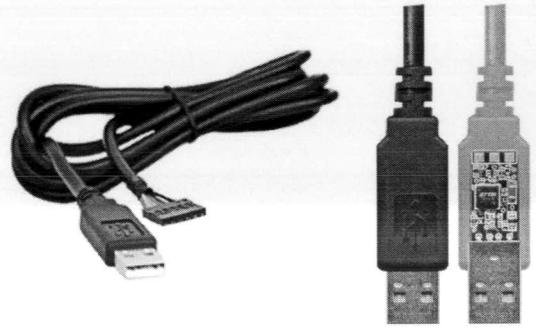


Figure 2.1: The TTL-232R is a USB-to-4BitIO port cable

I'll discuss HOW this works in a few pages time but, for now, let's WATCH it work. We learnt from Chapter 1 that all devices need a device driver – so download a driver for your operating system (OS) from www.ftdichip.com/FTDrivers.htm and load it on your PC; refer to Appendix A which includes instructions of how to do this for each supported operating system. FTDI has two sets of drivers and for the first few examples we need the D2XX driver so use that if your OS only allows a single driver for the FTDI device. Now plug the USB-end of the cable in to the PC; the OS may indicate that a new device is being added and it will match it with the device driver loaded in the previous step and this will be installed so that the OS can use it.

Figure 2.2 shows a schematic of our first example and shows this circuitry mounted on the first PCB. The button is connected on Bit3 which is pulled high by a 200K Ohm resistor inside the cable; this bit will therefore be read as a high unless the button is pressed when it will read as a low. The LED is connected on Bit2 and will be lit when this bit is driven high and will be off when this bit is driven low. Note that the resistors shown with dotted lines are included within the cable and Bit0 and Bit1 are not used in this example.

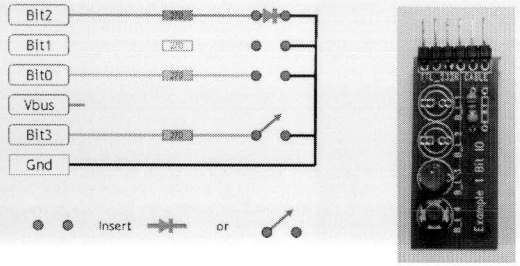


Figure 2.2: The first example, schematic and hardware

Figure 2.3 shows an edited version of the source code of our first example - I removed the error-checking for clarity but this is included in the supplied example1.cpp. Let me first explain the three helper routines, ***InitializeForBitIO***, ***WriteBits***, and ***ReadBits***, that will allow the main loop to be written more simply.

The OS enumerated the FTDI component in the cable when it was attached and it has been added to the OS-internal plug-and-play tables. The ***FT_ListDevices*** system call queries these plug-and play tables and returns a ***DeviceCount*** of matching FTDI devices which are currently attached - my code assumes that we only have one of these cables attached. The ***FT_Open*** system call gets a handle for this device that can be used in later system calls. The ***FT_SetBitMode*** selects which pins are input, which are output and sets operation to Synchronous Bit Bang mode. The ***WriteBits*** routine is an ***FT_Write*** of one byte to our device and the ***ReadBits*** routine is a similar ***FT_Read***. ReadBits returns the inverted value of the pins since a button press is active low.

```
BOOL InitializeForBitIO(void) {
    FT_CreateDeviceInfoList(&DeviceCount);
    if (!DeviceCount) return printf("No FTDI devices\n");
    FT_Open(0, &FT_Handle);
    FT_SetBaudRate(FT_Handle, 921600);
    FT_SetBitMode(FT_Handle, 4, SyncBitBang);
    return 0;
    }

UCHAR ReadBits(void) {
    UCHAR Value;
    DWORD BytesRead;
    FT_Status = FT_Read(FT_Handle, &Value, 1, &BytesRead);
    return ~Value;
    }

void WriteBits(UCHAR Value) {
    DWORD Written;
    FT_Status = FT_Write(FT_Handle, &Value, 1, &Written);
    }

int main(int argc, char* argv[]) {
    if (InitializeForBitIO() == 0) {
        while (1) {
            WriteBits(LED_Off);
            Idle(100);
            while (ReadBits() & Button) {
                WriteBits(LED_On);
                Idle(100);
                }
            }
        }
    FT_Close(FT_Handle);
    return 0;
    }
```

Figure 2.3: Edited source code of first example

The main loop polls the button every 100 ms and, if it is pressed, it will turn on the LED. The main loop will run until a Control+C is entered on the PC keyboard.

Let's run the program and watch it work.

Pause here while you run the program.

Now marvel at its simplicity.

So adding a push button and LED to the PC using USB wasn't difficult after all. If you don't like the gauge or the length of the cable you can just purchase the "plug + electronics" and add your own cable and case. The electronics supports 4 IO lines and these can be any combination of buttons and LEDs. Notice that you didn't see any descriptors or had to deal with any USB-ness at all.

How it works

The heart of the electronics, embedded within the plug of the FTDI cable, is an FT232R which is a self-contained, USB-ByteMover device. The block diagram, shown in Figure 2.4 shows the main elements of the FT232R.

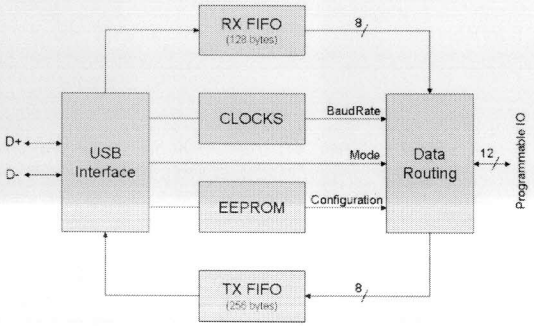


Figure 2.4: Block diagram of FT232R USB-ByteMover device

The ***USB Interface***, with the aid of data within the ***EEPROM***, enumerates with the PC host and selects FTDI's drivers. USB data packets are delivered to the ***RX FIFO*** and are then routed to the programmable IO pins using the EEPROM Configuration, Selected Mode and BaudRate generated from the ***CLOCK*** circuitry. The FT232R supports 3 data routing modes: Synchronous BitBang, Asynchronous BitBang and UART which supports RS232, RS422, and RS485 protocols with full modem control signals. The programmable IO pin block is expanded (twice!) and shown in Figure 2.5; each pin can be set as input or output, can be programmatically inverted and have higher drive current. An FTDI utility program, called FT_PROG (described in Appendix A) is used to set power-on parameters in the EEPROM. Similarly data can be routed from these programmable IO pins, including the UART protocols, to the ***TX FIFO*** where the FT232R collects this data, moves it to the PC and queues it ready for the FT_Read function. The FT232R handles all of the USB protocol on your behalf; bytes are moved between the PC application program and the programmable IO pins neatly and efficiently and the only reason to ponder about the actual data transfer is a concern about performance. In the examples in this chapter the performance bottleneck will be the IO speed at the programmable IO pins and this will be examined in later chapters. The performance limiter in this first example is the 10Hz human user – the USB operations are in the millisecond range and will be considered 'instantaneous' by the user.

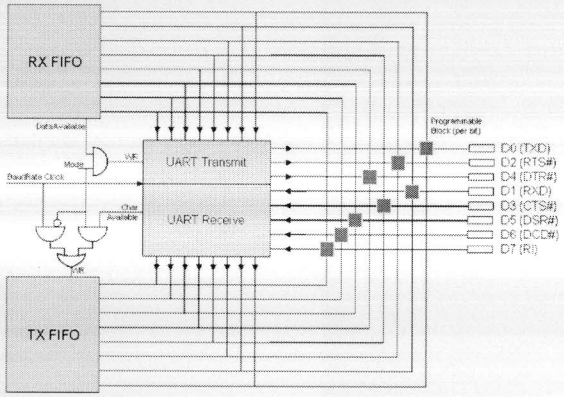


Figure 2.5A: Showing detail of ABUS data routing

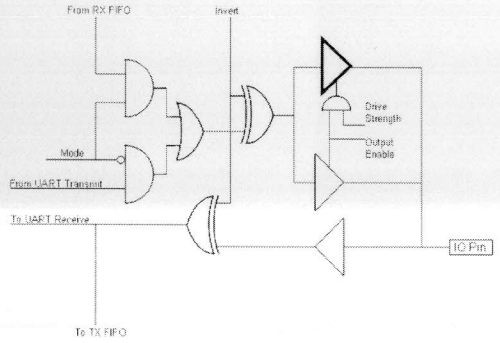


Figure 2.5B: Showing detail of programmable IO pins

Returning to the first example, we are using synchronous bit-bang mode so WriteBits copies the data byte into the RX FIFO which strobes data, at baudrate, directly to the IO pins. At the same time, the data on the IO pins is strobed into the TX FIFO and then delivered to the application program via ReadBits. Note that a WriteBits function is required before a ReadBits function can be used and note that they are paired to keep the read data in sync with the write data.

Getting more IO lines

Although the FT232R supports 12 programmable IO lines, only 4 are brought out on the TTL-232R cable. Our first example could be easily expanded to include any combination of up to 4 buttons and LEDs, and although it makes a great demo, it is a limited solution that can only solve a few problems. We need a solution that supports at least several bytes of IO. Figure 2.6 shows the schematic of a low cost component added to the non-USB end of the cable that will support up to 8 bytes of IO in any combination of inputs and outputs.

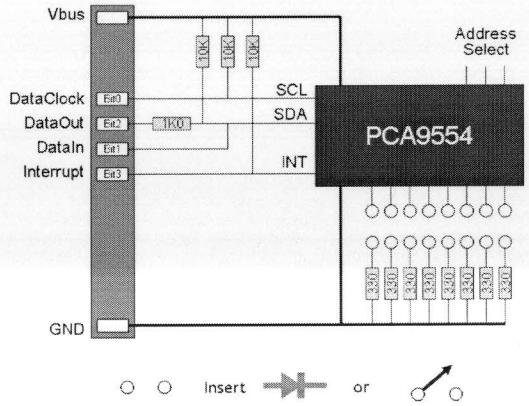
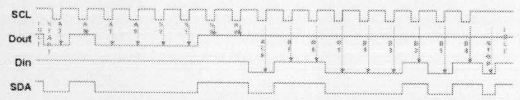


Figure 2.6: Adding an I2C IO expander to the cable

Rather than use the four programmable IO lines statically we are going to drive them with an I2C protocol and thus take advantage of the wide range of available I2C components. I chose I2C as an expansion bus since this is a multi-drop, 2-wire bus with a well-defined protocol that includes device addressing as well as data transport. This choice will allow me to easily expand the solution for later examples. I2C is a 2-wire bus but I need to use 3 connections from the cable since Output and Input functions are on two separate pins. One pin is used as ***DataClock*** or ***SCL***. I must keep my ***DataOut*** pin high when the I2C device is driving ***SDA*** low so that this can be detected by my ***DataIn*** pin.

Let's first consider the case where I have eight buttons connected to the PCA9554 component. Let's also assume that I have pre-selected register 0 using a command byte write such that an I2C read command will read the input pins. I have redrawn figure 10 from the PCA9554 data sheet as my Figure 2.7 to show the waveform that must be generated. In particular, I have separated out the wired-OR, I2C SDA line into my DataOut and DataIn lines so that it is easier to see who is driving this shared line.



Blue: FT232R driving SDA, Ax = PCA9554 Family Address, Sx = Sub Address
Red: PCA9554 driving SDA, Dx = Input data = 0xC5

Figure 2.7: Waveform needed to read an I2C byte

As you can see, each I2C bit transition needs three byte writes so, with an eight bit command, one bit ACK, eight bit data read, and a one bit STOP this will result in 54 bytes that need to be written to the FT232R. Example2 calculates this byte stream at run time and sends it to the RX FIFO where it is clocked out at the selected baudrate. The PCA9554 can operate at up to 400 KHz and the baudrate must be chosen to meet the minimum timings of their part. The limiter is a clock low time of 1.3µs which means a baud rate divisor of 4, or 5 with some margin.

The PCA9554 sub address is also calculated at run time and this three bit address field allows up to eight of these components to be used. This results in an easy expansion up to eight bytes of digital IO. Figure 2.8 shows example 1 built on a solder-less breadboard. There are many products available but I use the range from Elenco Precision since they are built from modules that may be reconfigured to give a better layout area. Figure 2.8 shows two PCA9554's, one with buttons implemented with a DIL switch and one with LEDs implemented with an LED bar graph.

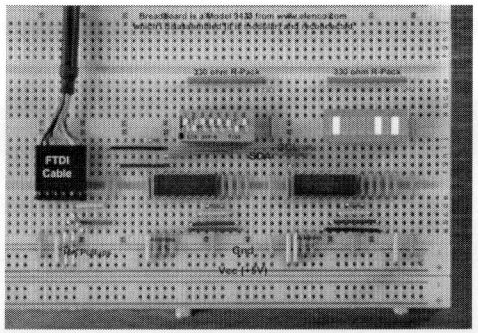


Figure 2.8: 2-way I2C bus expansion using PCA9554

Figure 2.9 shows a hardware variant of example 1. The Microchip MCP23008 has the same capability as the PCA9554 (actually, it has a lot more, but my example does not use this) and a better pin-out for this example; it allows 4 ports in about the same area and I chose 4 large seven-segment displays. You could easily add 4x8 = 32 buttons to this example, enough for most control panels.

The breadth of I2C components also allows us to have analog in and analog out modules using the Analog Devices AD799X or AD53X1 for example. These boards could be used standalone or could be used in conjunction with the buttons and LED boards. There are also sophisticated ICs such as TV tuners, sound processors and a wide range of multi-media circuits available with an I2C control interface. So, with this cable and a few standard components I can simply access up to 64 bits of digital IO and several analog channels -

enough for many control panels, system configuration, or even a distributed data gathering system. We are using PC software to bit-bang an I2C interface and this is implemented in a modular, expandable program called Example2.cpp.

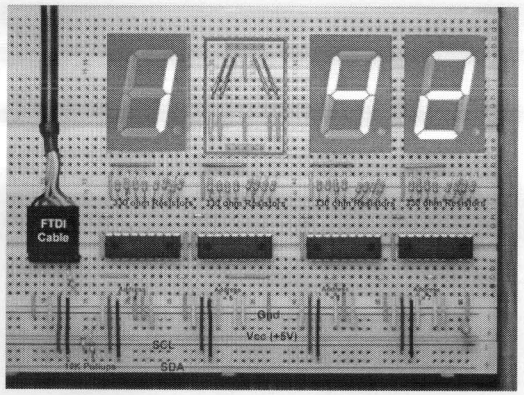


Figure 2.9: 4-way I2C bus expansion using MCP23008

Chapter Summary

This chapter has shown that is it easy to attach simple IO to a PC using USB. We used an FTDI TTL-232R cable to first drive discrete buttons and LEDs then, with the help of some low cost I2C components, we connected up both digital IO and analog IO to a PC. The USB cable also supplied the power source for our components. I have this up and running and I didn't have to even open the USB spec! All the USB-ness is handled by the FTDI device and the device driver, allowing me to concentrate on my application.

Chapter 3 - Serial and parallel device conversion

The embedded world still uses serial ports and parallel ports because they are easy, especially when compared with USB! A serial port uses just two data wires, TX and RX, for full duplex communications and a reference ground wire is also essential. Serial ports became popular with the introduction of modems in the 1977 and the RS232 standard also includes modem control signals such as DataSetReady and DataTerminalReady. The signaling levels are +/- 12 V and this tends to limit the maximum data rate to 56KBaud. Once both ends of the wire agree on the baud rate it is a simple matter to send and receive any stream of data bytes.

Unfortunately the simplicity of exchanging any stream of data bytes is also the serial ports Achilles heel. Most serial links also need to exchange some control information and this is embedded in the data stream using some kind of escape sequence. This, by itself, is not a bad technique - the issue is that there is no standard escape sequence which results in applications software being tied to a specific piece of hardware. Again, this is not a major problem except that there is no standard way for the application software to identify the attached hardware.

The real problem comes when you want to attach your serial device to a PC and you discover that there are no serial ports! PC hardware changes more rapidly than a typical embedded system, and if we are to take advantage of 'PC economics' then we need to follow their trend.

PC software has also changed dramatically. The first PCs introduced by IBM were well documented and all of their internal hardware was exposed via BIOS listings. You were actually encouraged to access the serial ports at 0x3F8 and 0x2F8. This all changed with the introduction of 'protected mode' Windows where applications software was prevented from accessing the physical hardware. The same is true today about Mac OS X and Linux. All three operating systems support multi-tasking and multi-applications so they must own the underlying PC hardware so that they can manage its use. The impact to the embedded developer is that the serial ports must be accessed via a device driver - in Windows this is COMxx, and in Mac OS X and Linux, this is /dev/tty. Once you encourage the OS to supply a *handle* to a serial port then you can read and write streams of data bytes from and to the serial port.

Representative Serial Device

So let's work through an example of converting a serial device to a USB device. My representative device is a serial display from www.robokits.co.in, as show in Figure 3.1. You can download a user manual from their web site but it is basically a 2 line by 16 character display that accepts ASCII characters through a 9600 baud serial connection. Non-displaying characters (0x00..0x1F and 0x80..0xFF) are interpreted by the on-board micro-controller to implement special functions such as the setup and display of custom characters and turning the backlight on and off. I have used this display on many embedded projects since it is low cost and only needs a single IO pin to drive the display. Adding it to an embedded PC would be much cheaper than a VGA display for those applications that only needed 2 lines by 16 characters or it could be a remote display in addition to the main display.

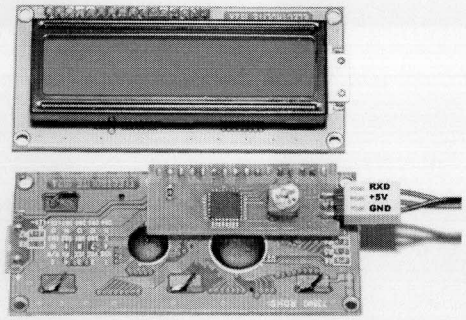


Figure 3.1: Representative Serial Device

Rather than create a custom example program for this chapter I thought it more convincing to use software for the PC that is already available and designed to support serial ports. For the Windows PC I shall use ***HyperTerminal*** and for the Mac/Linux PC I shall use ***CoolTerm*** (download from http://freeware.the-meiers.org/). Using the same TTL-232R cable introduced in chapter 2, first connect the non-USB end of the cable to the display as shown in Figure 3.2. We are using the cable to power the display and have TXD looped back to RXD.

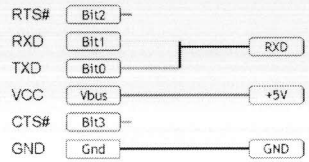


Figure 3.2: Connecting the FTDI cable to the display

Windows PC operation:

The FTDI drivers that we installed in chapter 2 includes two distinct interfaces, D2XX which we used to access low level functions and VCP, a Virtual Com Port interface. The windows driver supports both interfaces in a single installation, called CDM, but only one may be used at a time. If you have not installed this driver, do it now.

Insert the TTL-232R cable into your PC and then display the hardware configuration using the Device Manager in the system control panel. My configuration is shown in Figure 3.3. Note that the cable enumerated as a COM port - mine happened to be assigned as COM11 and yours will probably have a different number.

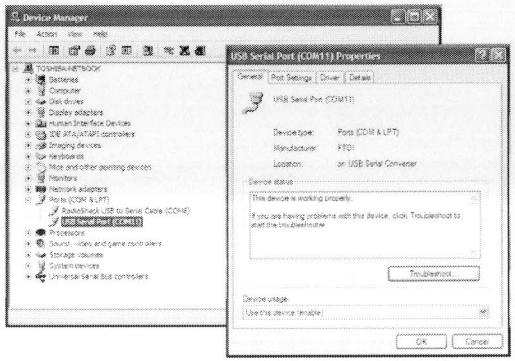


Figure 3.3: The cable is recognized by Windows as a COM port

Spin up HyperTerminal in the accessories directory and select the COM port that has been assigned to the TTL-232R cable. Now jump to the 'Configure the terminal program' section on the next page.

Mac Operation

Life is a little trickier for the Mac user since the FTDI drivers are not combined. We installed the D2XX driver in chapter 2 and it is now time to install the VCP driver. Decompress the FTDIUSBSerialDriver.dmg file that was downloaded from FTDI's web site and click on FTDIUSBSerialDriver package and follow the installation instructions.

Insert the TTL-232R cable into your PC and the OS will preferentially choose FTDI's VCP driver. Figure 3.4 shows the output of the System Profiler tool and, as seen, it lists the FTDI cable connected to USB.

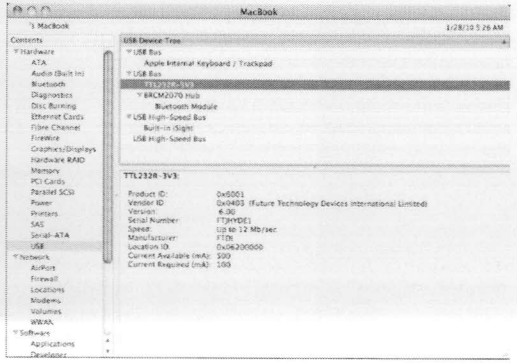


Figure 3.4: The cable is recognized by Mac OS X as a COM port

Now spin up CoolTerm and select the usbserial device.

Configure the terminal program

Choose 9600 baud and open a connection. Type "Hello World" then note that this also appears on the 2 line display. We're basically done! Yes, we are using USB but it is embedded. In fact, it is embedded so deeply that we haven't even been exposed to any USB at all. All of the USB aspects have been handled by the cable and by FTDI's VCP driver.

So conversion of a serial device to a USB device is almost trivial if we use this TTL-232R cable and driver from FTDI. All of the hard work is being done by the operating system and its drivers - they are handling the differences in hardware and we, at the application program level, need not be concerned about exactly how this is being accomplished.

Switching back to the D2XX driver.

The D2XX driver is always available to the Windows user so you may skip this section. The Mac OS X user can temporarily or permanently remove the VCP driver as follows; use the Terminal application and view the system extensions to identify the system name of the FTDI cable:

```
cd /System/Library/Extensions
ls
```

It will probably be `FTDIUSBSerialDriver.kext`. You can remove it for the current session with:

```
sudo kextunload FTDIUSBSerialDriver.kext
```

To permanently remove it (which will mean reinstalling the package if you wish to use the VCP driver again) use:

```
su
rm -R FTDIUSBSerialDriver.kext
```

Optimizing the serial connection

Now, before you rush out and make a volume purchase of this cable let us look at a few optimizing steps. My serial display is not typical in that it used TTL levels rather than RS232 voltage levels. The top of Figure 3.5 shows a more typical serial device. It has an internal microprocessor or microcontroller that drives an RS232 voltage converter for PC communications and drives custom IO specific to the embedded application.

In the center diagram of Figure 3.5 I have replaced the serial cable with the FTDI cable. This cable drives TTL levels so there is no need for the RS232 voltage converters. We have a problem with the connector however since the industry expects RS232 voltage levels on the 9 pin (or 25 pin) serial connector. I shall deal with this issue in a moment.

Now look at the third diagram in Figure 3.5 - I have moved the FT232R part from the "PC end" of the cable to the "device end" of the cable. This FT232R part replaces the RS232 voltage converter and I replace the serial port connector with a USB B connector (standard size or mini-B). This means that I use a standard USB cable to connect my new device to the PC. Another advantage of having the FT232R at the 'device-end' of the cable is that you have access to all 12 IO lines. We shall look at these in the next chapter.

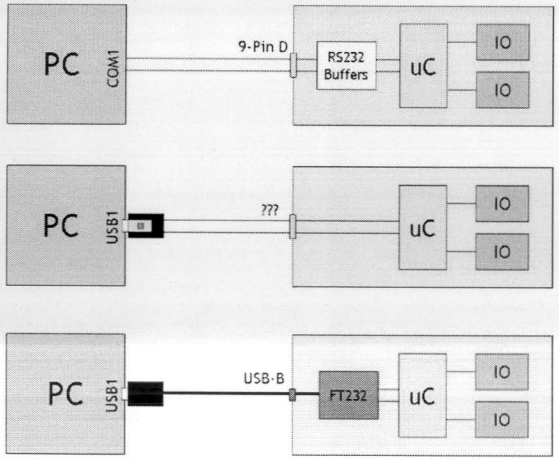


Figure 3.5: Converting a serial device

Total conversion effort is less than a day. We migrated a serial device into a USB device. But what else did we gain besides a product that is likely to sell better?

If needed, we could increase the baud rate to the device. Standard serial cables can easily support 56 Kbaud and some can do 192 Kbaud. The FT232R can run at 3,000 Kbaud due to the higher data transfer rate of USB. If your device moves a lot of data then this "upgrade" would be worth implementing.

A USB cable can also supply up to 500mA at 5V. If your device can operate at or below this power level then you could eliminate the power source from your device and thus reduce its manufacturing cost. And since you will charge more for a USB version then you get a double cost benefit as well as a simpler product. This is also "low hanging fruit" and is easy to implement.

Converting a parallel device to USB

FTDI have a trio of "USB-ByteMovers" that can be used in this type of application. So far we have been using the FT232R that has a serial interface. A companion part, the FT245R replaces this serial interface with a parallel bi-directional FIFO interface for higher data throughput rates. Converting a parallel interface device to a USB device follows the same methodology as the serial device. From an applications software perspective you still treat it as a serial port but otherwise the software is un-changed. You can also reap the higher speed, and USB-provided power, benefits of the serial port conversion example. A dual-channel part, the FT2232D, is also available: the two channels can be individually programmed to operate as an FT232R or an FT245R or they can be combined to produce higher capability interface.

Chapter Summary

In this chapter we used FTDI's VCP driver that presented the USB device as a COM port. While this is convenient as a transition strategy it does not address the serial ports Achilles heel of device identification. If you have multiple devices and chose the wrong COM port number then your application software will fail, just like it did in the olden days when using real COM ports. The VCP driver does such a good job of hiding the underlying hardware details that some necessary information for a multi-device system is also hidden. I would recommend using the VCP driver as an initial step but migrate to the D2XX driver in the longer term since it has more capabilities.

Each FTDI component has an integrated, or attached, EEPROM that includes a unique ID programmed during the manufacturing process. A custom Vendor ID (VID), Product ID (PID) and friendly name can also be programmed into this EEPROM (Using FT_PROG, described in Appendix A) and any combination of these parameters can be used to specify which particular device of a Multi-USB device system should be opened by an application program.

Now that we know how easy it is to have multiple devices the next chapter will look at more capable devices.

Chapter 4 - Connecting to more capable devices

Chapter 2 showed that the bit-banged IO pins of the FT232R could be used to create a 400KHz I2C expansion bus and enabled this component to solve a wider set of digital IO and analog IO problems. FTDI has taken this fundamental IO expansion concept a major step forward and integrated two FT232Rs with more capability and bigger FIFOs into a USB high speed FT2232H product, a block diagram of which is shown in Figure 4.1. Note that the FIFOs are 32 x bigger! The I2C bus generation, including a more efficient parallel to serial conversion, has been added as a mode called MPSSE (Multi-Protocol Serial Synchronous Engine which also supports SPI, JTAG and any custom protocol). Also added are an 8051-type bus emulation and a fast, opto-isolated serial protocol. Each of these two interfaces can independently run a synchronous protocol up to 30MHz or a serial protocol up to 12Mbaud. There are also digital IOs that can be bit-banged. Let's see how this improved part can solve a wider variety of design problems.

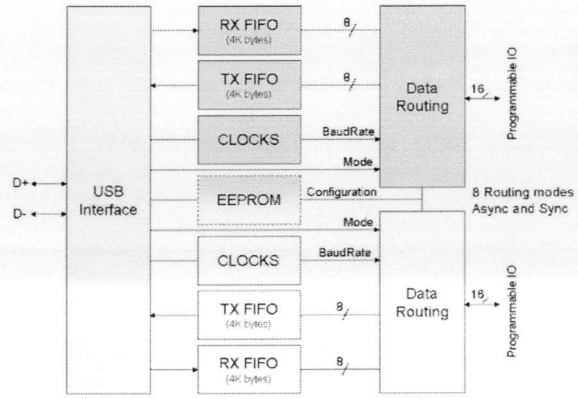


Figure 4.1: Block diagram of FT2232H

Data Collection Pod

Figure 4.2 shows a block diagram of a "Data Collection Pod." This pod is battery powered and is physically small and light to enable it to collect data from a wide range of sources. Once enabled it collects data from three analog sensors and stores these data samples in an 8MB Atmel DataFlash. At the end of the data collection period the pods are connected to a PC to extract the data and to recharge the lithium battery.

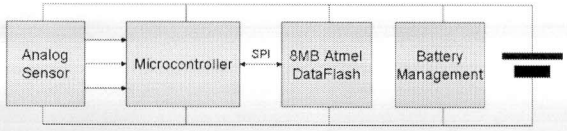


Figure 4.2: Block diagram of data collection pod

There are many applications that have a similar block diagram. In this application data is being collected but, with transducers rather than sensors, this block diagram could also be a data distribution system. My point here is that I am describing a general application using a specific example.

The obvious method of connecting the data pod to a PC is via USB. This will mean choosing a microcontroller that has a USB interface or by adding a USB component such as the FT232R as shown in Figure 4.3. Since we have a lot of data to move perhaps we should consider high speed USB. Both options increase the size, weight, and current consumption of the data pod so we look for a more creative solution.

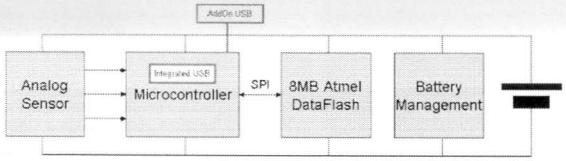


Figure 4.3: Options for adding USB

Figure 4.4 shows an optimized solution that partitions the design into a "Reader" and a lower-cost "Data Pod". The data pod connects to the reader using the SPI connection on a set of PCB gold fingers - this saved the size, weight and cost of a connector and did not involve additional circuitry in the data pod. Additionally the battery management IC was moved out of the data pod and into the reader since it is only needed during charging.

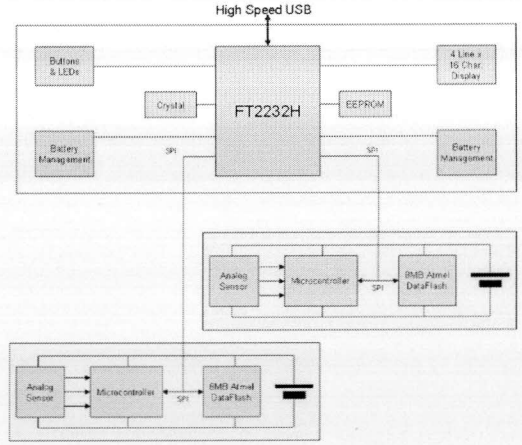


Figure 4.4: Block diagram of 'Reader' plus 'Pods'

The FT2232H can support two USB-to-SPI channels and run them both at 30MHz. There are also enough additional IO lines to manage two battery management ICs, support a series of buttons and LEDs and a 4 line by 20 character display that can give the user instructions or sales messages. This means that the "Reader" is a standalone peripheral that does not need the PC screen or keyboard to implement a human interface. So, if needed, a single PC could support several of these readers. Lets step through this example which I have modularized to create a set of easy-to-adapt building blocks for your use.

Figure 4.5 shows a more detailed block diagram of the IO connections to the FTDI FT2232H channel A which will be set up in MPSSE mode. Channel B is similar but has buttons and LEDs in place of the LCD display. I will be going into detail on the SPI interface to the Atmel AT45DB642D 8MB DataFlash and on the custom parallel interface to the LCD display. For prototyping I used the FT2232H mini module, shown in Figure 4.6 which I wired to the DataFlash and to the display. All components are powered from USB.

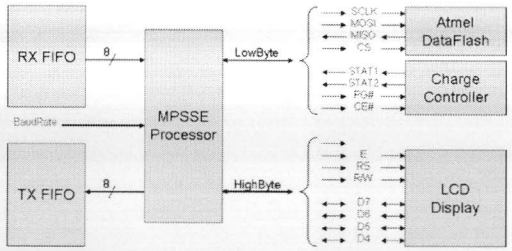


Figure 4.5: Detail of FT2232H Channel A IO connections

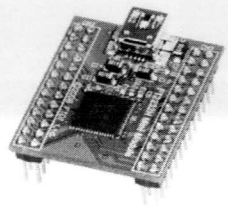


Figure 4.6: FT2232H mini module used for prototyping

SPI interface

In MPSSE mode, command bytes are intermingled with data bytes within the RX FIFO and the MPSSE processor decodes these command bytes and operates on any data bytes - this can be a little confusing at first so I shall step through this SPI example in detail. Rather than repeat a lot of information in FTDI's applications note AN_108. I recommend that you have a copy of this note to refer to as I work through this example.

The MPSSE command structure enables data to be strobed out of the RX FIFO at a bit or byte level on the rising or falling edge of the clock. Data can also be strobed into the TX FIFO with similar control. Our first task then is to choose a signaling method that is compatible with the Atmel DataFlash. Referring to figure 21.1 of the AT45DB642D data sheet we note that in SPI mode 0 SI data is latched on the rising edge of SCK and SO data is driven on the falling edge of SCK. We therefore set up MPSSE to drive byte data out on the falling edge of SCK (i.e command 0x11 from AN_108, Table 3.3) and to read data on the rising edge of SCK (i.e command 0x20 from Table 3.3). The Atmel DataFlash requires CS to toggle to initiate commands and I will use SetDataLow commands (command 0x80 described in section 3.6 of AN_108) to drive CS low and high.

Let's first read the Device ID from the DataFlash. After driving CS low we need to send a command byte, 0x9F (See table 15 of Atmel data sheet), we then read in 2 bytes and finally drive CS high. This sequence is shown at the left hand side of Figure 4.7.

Drive CS Low	=	0x80	0xF6	0x0B	
Send Command Byte	=	0x11	0x00	0x00	0x9F
Read 2 Response Bytes	=	0x20	0x01	0x00	
Drive CS High	=	0x80	0xFE	0x0B	

Figure 4.7: MPSSE commands used to drive SPI

A 3 byte sequence is needed to drive CS low and this is shown on the right hand side of Figure 4.7 - this SetLowByte sequence can set up to 8 bits. 3 bytes are needed to send the SPI command byte - bytes 2 and 3 are a count of the following data bytes, and, in this example, there is only one byte (0x0000 = 1 byte). This may appear to be a large overhead but the PC is running very fast and the FIFOs are large so you should not be overly concerned about this. Since count can be up to 64,536 the overhead is less for larger data transfers. 3 bytes are needed to set up the read of the response from the DataFlash. Finally 3 bytes are needed to drive CS high which will return the SPI bus to its idle state.

So, we load the RX FIFO with 13 bytes and execution of these commands by the MPSSE engine will result in 2 bytes will be written into the TX FIFO. Figure 4.8 shows the resulting SPI traffic captured with a USB DX logic analyzer (see www.usbee.com). I added an extra chip select and deselect, so that we could get a little more insight into the sequence timing. I have the baud rate set to 1 MHz during debug and I will run at 30 MHz later.

Refer now the Example3.cpp program listing - I have several helper routines that allow you to focus on the SPI operation and not on the details of the MPSSE implementation. Most DataFlash commands are 4 bytes long so I declare them as 4 byte dwords and manage the individual bytes inside the helper routines. I have implemented GetDeviceID, ReadData, and WriteData to get you started. Note that the bit-bang instructions (SetLowByte used to toggle CS) cycle the IO pins at the selected baudrate.

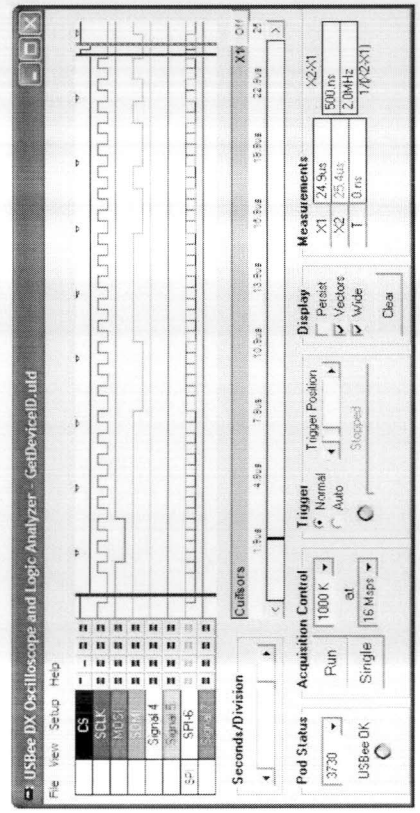


Figure 4.8: USBee trace of GetDeviceID SPI command

LCD Interface

Most 2 and 4 line character displays use the same parallel interface consisting of 3 control lines (E, R/W, and RS) and an 8 bit data bus that can be operated in 4 bit mode. In this application I have only 8 data lines available (called GPIOH0..7 when in MPSSE mode) so I implemented the data transfer in 4 bit mode. The waveforms needed to write and read the display are shown in Figure 4.9. I extracted these from the LCD display datasheet which is included in the Example 4 directory for your convenience. I use a series of SetByteHigh commands to create this custom waveform. To meet the LCD displays timing I set the baud rate to 1 MHz when writing to this high data byte.

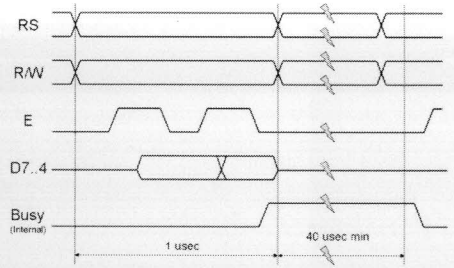


Figure 4.9: Control signals used by most LCD character displays

Sending a command to the LCD display takes about 1µs and then the display needs at least 40µs to implement the command. Some commands take much longer. Please refer to the LCD display datasheet. The datasheet also recommends polling the status register looking for a busy bit but we will NOT do this due to the fundamental operation of USB. It is time to understand the latencies involved with USB transfers.

Sending data to the RX FIFO involves an FT_Write command and reading from the TX FIFO involves an FT_Read command. If two commands are initiated (two FT_Writes or FT_Write + FT_read) then the OS will schedule these in separate USB frames. In other words, they will be at least 1ms apart. So it is not sensible to poll for a 40µs signal since it will take 1ms to do the poll! It is also a good idea to send as many bytes as possible in a single buffer; otherwise the

separate FT_WRITEs will be 1ms apart. The FT2232H has a 4KB RX FIFO and can therefore queue up a large number of commands + data for the MPSSE engine. In this example I idle for 40µs between most LCD commands using the MPSSE command 0x8F, 0x38, 0x00.

My simple example, within the Example4 directory, allows you to send any text to any line. The frame work is complete and other functions, as described in the LCD display datasheet, may be easily added. I tested the code on several displays of different physical sizes, some 2 line some 4 line, and they all operated the same way.

For the Data Collection Pod example I have SPI on both channels and I bit-bang the battery management ICs, LCD display, buttons and LEDs. The FT2232H makes an excellent dual USB-to-SPI adaptor with additional capability to read and write 24 additional IO lines.

Other examples

The MPSSE mode of the FT2232H supports SPI, I2C, JTAG and custom parallel protocols on both channels. So, as an exercise, we could restructure the FT2232H's channel B to drive the I2C protocol and run the examples from chapter 2. Or we could reprogram the FT2232H's channel A to be a serial interface and run the examples from chapter 3. The FT2232H also supports several other modes and protocols that I have not presented here (look over the datasheet!) making it an extremely versatile component suitable for many interfacing projects.

Figure 4.9 shows an alternate hardware implementation for a single channel DataPod using a solder-less breadboard. I used a DLP-1232H module since this DIL module plus straight in! A downside is that only part of a Channel A is brought out to the module pins; high byte is not brought out so I implemented the 7-pin LCD interface on an MSP23S08 expansion device (this is an SPI version of the MSP23008).

I hope that I have given you a flavour of the capability of the FT2232H - it can be any two of:

USB-to-SPI adaptor USB-to-I2C adaptor
USB-to-JTAG adaptor USB-to-custom protocol adaptor
USB-to-serial adaptor (RS232, RS422 or RS485)

And if you need four channels then look at the FTDI FT4232H.

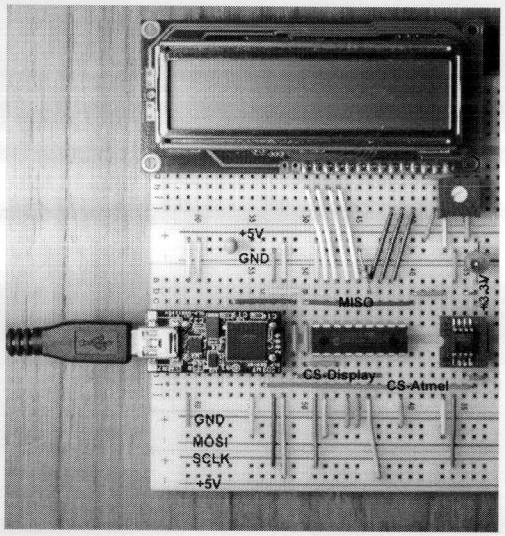


Figure 4.10: Single channel DataPod using a DLP-1232H module

Chapter summary

Notice that we write software on the PC to create our industry-specific device. The modes of the FT2232H allowed us to create a variety of solutions with a single component which is configured using software. There is no firmware to write or maintain at the USB device since the FT2232H is implemented as a fixed-function, high-speed device. The drivers hide all of the details of USB and we can focus on filling the RX FIFO and reading the TX FIFO – this enables us to be closer to our application.

Introduction to Part 2

Developing Embedded USB Host Controller Applications

We learnt in part 1 of this project book that most of the complexity of a USB system is within the host controller. The host controller is responsible for managing the communications to a diverse range of USB devices and this must be scheduled using predefined rules and many timing constraints. Early USB host controller implementations used a fast RISC CPU that was tuned to process the various transaction lists and to handle the required error checking and retries. FTDI took a different approach with their Vinculum-II host controller - they implemented most of the host controller functions in dedicated, special-purpose hardware such that the host controller function could be managed by a simpler 16-bit CPU. There are still timing constraints but these are handled by a real-time micro-kernel (which is described in later chapters). This 'hardware-heavy' implementation results in a USB host controller that is easy to use and is the main subject of part 2 of this project book.

FTDI provide more than just the silicon components; their full solution includes a complete C-based tool chain with a GUI-based Integrated Development Environment (IDE), a Real Time Operating System (RTOS), an on-chip debugger and evaluation modules. There is a lot of material to cover! Chapter 5 looks at the original Vinculum host controller and its applications range –Vinculum-II can do everything that the original Vinculum can do plus a lot more! Chapter 6 looks at the hardware capabilities of Vinculum-II and chapter 7 looks at the micro-kernel and device drivers wrapped around this hardware. Chapter 8 works through a design example using the Vinculum IDE and I round off part 2 with several worked design examples. You will discover that developing a product that requires USB host capability is a straight-forward, well-defined task – you will call it "easy" after the second project!

Chapter 5: Vinculum-I Design Examples

FTDI introduced their first generation dual USB host controller, now called Vinculum-I, in 2006. It is a fixed-function device that supports the attached mode of operation as shown in Figure 5.1.

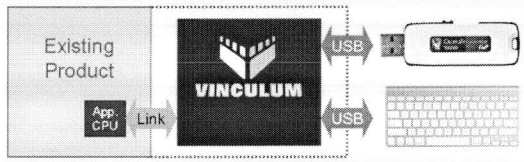


Figure 5.1: Vinculum-I operates as an attached device

Vinculum-I runs a firmware monitor that is controlled by an external application CPU using an SPI, FIFO or UART link. Several firmware versions are available that implement a variety of specific functions but all include the ability to read and write to a USB flash drive. This chapter looks at several examples of attaching a flash drive to an existing product using Vinculum-I.

Adding a Flash Drive to a product

The flash drive is arguably the most successful USB product. Its density has increased almost logarithmically over the past decade while its price has fallen at a similar rate. You can now buy 1GB drives for less than $10. But, up until now, they have been excluded from embedded projects due to the complexity of interfacing but I am about to change all that!

The major issue is, of course, that a flash drive is a USB device and therefore, to control it, you need a USB host controller. The USB specification deliberately put most of the communications complexity within the host controller, since there is only ever one in a system, and this enables USB devices to be simpler and therefore lower cost. A flash drive is a Mass Storage Class device and, although these specifications are a free download from www.usb.org, they are not easy to read. This is not surprising because they are specifications and not implementation guides. Additionally, these Mass Storage Class specifications only define basic track/sector, read/write operations so we also need to understand specifications of the FAT file system, as used on all commercial flash drives, to be able

to read and write user data. The amount of information that we need to understand how to "just connect a flash drive" is becoming over whelming. What we need is a component that implements all of these specifications for us; after all, they are industry standard specifications that we have almost no freedom to change anyway, we just want to use them!

Vinculum-I provides a DOS-like, command line interpreter, front-end to a flash drive. A Vinculum block diagram is shown in Figure 5.2 – internally it is implemented as a microcontroller, with specialized IO devices, running embedded firmware but we do not need to know this. Vinculum-I's command line interface is accessed via a UART, SPI, or a FIFO. Vinculum-I actually supports two USB ports and each can be programmed to be a host or a device but my series of examples will assume a single host port with a connected flash drive.

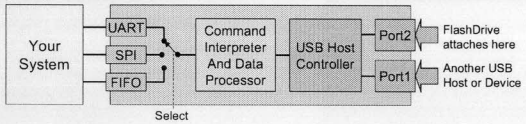


Figure 5.2: Vinculum-I uses a DOS-like command interface

To demonstrate its ease of use I am going to connect the FT232R USB-to-serial cable introduced in Chapter 2 to a PC that is running HyperTerminal at 9600 baud. I will connect this cable to an FTDI evaluation module called VMusic as shown in Figure 5.3. Ignore the name for now, we won't use the "music" part until the second example; for now, this is a Vinculum-I mounted on a board with a convenient serial connector. Choose any flash drive that you may have and plug this into the USB A socket of the VMusic board, also shown in Figure 5.3.

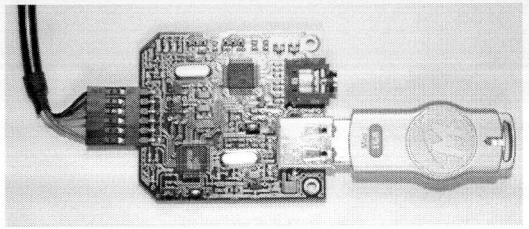


Figure 5.3: USB-to-Serial cable connected to VMusic board

The board will sign on and offer a D:> prompt. Now, in HyperTerminal, enter "DIR" and, Hey Presto, the contents of the drive are displayed. Now enter the following commands:

```
OPW test1
IPA
WRF 12
Hello World!
CLF test1
```

These commands first opens a file called "test1" for write, then tells Vinculum that 12 bytes of data are coming. "Hello World!" is the data that is written, and CLF closes the data file.

Now remove the flash drive and connect it to your PC, Mac or Linux system and open test1. Notice that the data written by the Vinculum is present. Now edit test1 to add a message "Hello from my PC, Mac or Linux"

Now reattach the flash drive to the VMusic board and enter "RD test1". Voila, the text is displayed!

Now stop and think what we have accomplished.

We have written, read and exchanged data files between a PC, Mac or Linux system and an embedded system using a flash drive. We did not have to learn USB, the Mass Storage Class specification or even the FAT file system. It was as easy as entering DOS-like commands on a serial connection.

Pretty amazing!

Vinculum-I powers up in Extended Command mode where all the commands and data are ASCII; some of these commands are summarized in Figure 5.4. It can be switched into Short Command mode where binary commands and data can be exchanged. The VMusic board only provides access to the UART connection but this will be enough for my first set of examples. Vinculum-I is also available in an OEM 24 pin DIP and this additionally provides access to the SPI port, the parallel port FIFO and the other USB port.

	Directory Operations
DIR	Lists the current directory
MKD <name>	Creates a new directory <name> in the current directory
DLD <name>	Deletes the directory <name> from the current directory
CD <name>	The current directory is changed to the new directory <name>
CD ..	Move up one directory level
	File operations
RD <name>	Read file <name>. This will return the entire file
OPR <name>	Opens file <name> for reading with 'RDF'
RDF <size>	Reads <size> bytes of data from the current file
OPW <name>	Opens file <name> for writing with 'WRF'
WRF <size>	Writes <size> bytes of data to the end of the current open file
CLF <name>	Closes file <name> for writing
DLF <name>	This will delete the file from the current directory and free up disk space
VPF <name>	Play an MP3 file. Sends file to SPI interface then returns
REN <n1><n2>	Rename a file or directory
	Management Commands
SCS	Switch to the short command set
ECS	Switch to the extended command set
IPA	Input data values in ASCII
IPH	Input data values in Hex
SUD	Suspend the disk when not in use to conserve power.
WKD	Wake Disk and do not put it into suspend when not in use
SUM	Suspend Monitor and stop clocks
FWV	Get Firmware Versions
FS	Returns free space in bytes on disk

Figure 5.4: Some of the monitor's DOS-like commands

There are two types of project suitable for an attached Vinculum device – data distribution and data collection and I have examples of each category. Typically, data to be distributed is created on a PC using specialist tools and then copied onto a flash drive; an embedded system then accesses this information and presents it to a user or a machine. My example is a small JPEG viewer and MP3 player – something that we would take on a business trip and that plays back images and sounds of our family, or our favorite music. If I had used a larger display I would have called this an "active photo frame" (it is on my TODO list!). My data collection example is a portable data logger that collects field data for later analysis by a PC. In both cases an application microcontroller is used to drive the Vinculum-I (since it is a peripheral device) and other circuitry. I am confident that you can dream up many more applications for this easy-to-use part.

JPEG viewer and MPEG player

I chose a Cypress PSoC for the application microcontroller since it has firmware-configurable hardware that allows me to solve a wide range of problems with a single device. I develop and debug using a "high-end" PSoC device that has ample analog and digital resources then, near project completion, I can select a lower cost device within the same family. For the first example I shall use the Cypress PSoC Evaluation board and this is shown in Figure 5.5 with the VMusic board and a 1.5" x 1.5" Micro-LCD display already attached to the breadboard area.

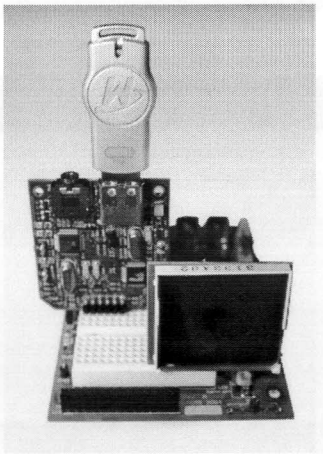


Figure 5.5: This example was developed and debugged using a PSoC development system

Fundamentally I have a PSoC that reads image files off a flash drive using commands sent to the Vinculum monitor, the PSoC then sends this image to the display. If a matching MP3 file is also present on the flash drive then I command Vinculum-I to play it – this could be music or a person talking. A PC is used to create the image and MP3 files and these are copied onto the flash drive. The

PSoC/Vinculum-I based player then "runs the show." A JPEG viewer and MPEG player is the base example but an interactive display that could be used in stores, museums, product demonstrations, art galleries, etc. is a straightforward design extension. A series of flash drives in English, Spanish, Japanese, etc. could be used to create a more universal solution.

Another beneficial aspect of a PSoC-based design is that Cypress has over a hundred applications notes that describe building blocks that can be used within your own design. The PSoC could scan buttons and the application program would use these button inputs to navigate through images / MP3 files. Or the PSoC could support a touch screen using a few of its configurable analog and digital blocks. This could be a simple resistive screen overlay or a more reliable CapSense implementation.

The size of the graphical display determines the complexity and choice of the PSoC. Since this example is about embedded flash drive applications I chose the simplest display to implement here and I will cover interfacing to a larger display in a future update. I found a serial-interface, micro-LCD and was very impressed with ease of use of these 128x128 color displays. These 1.5" x 1.5" displays are not expensive – you should get some and experiment with them. I am sure that you will soon find many uses for them, just as I did. I personally found the OLED displays much better to look at when compared with the LCD displays but the firmware to drive both displays is identical. The micro-LCD module is a very capable subsystem that supports graphics rending and several fonts. My example uses about 5% of its capability as I just download images to it. These images are 128x128 by 16-bit color and I use a PC application called Graphics_Composer that converts JPEG, BMP, and GIF images into this format (this is included in the download package). In this example these images will be copied to a flash drive and called nnn.img (nnn = 000 to 999). MP3 files are also created for each image and they will be called nnn.MP3 (these could be your favorite songs renamed).

From an application software perspective we have a PSoC interfacing two serial connections, a Vinculum-I and a micro-LCD connection. The application starts by looking for 001.img and copies it to the display. If it finds 001.MP3 then it will play it, else it will wait for 60 seconds (easy to modify) before moving onto 002.img. The application keeps incrementing through filenames until one is not found then it starts at 001.img again. To change the photos and/or

music you just swap the flash drive. The complete PSoC project is downloadable from my website and, as you will see, it supports the basic function. It is easy to expand this design to add functions – I plan to add a feature-rich alarm clock once I get some spare time. It would be easy to make this battery powered however displays tend to consume a lot of power so I would also add a battery charger in this case. A battery charger uses a few analog and digital resources of a PSoC, a few FETs, an inductor and R's and C's. This design extension is covered in detail in Cypress's application note collection.

Portable data logger

I was "persuaded" to create an example based on the PIC microcontroller. I personally didn't like this part due to its "weird" instruction set. However, my colleague Don Powrie of DLP Design introduced me to the CCS toolset and these make PIC designs actually pleasant to do! I used to be a staunch advocate of only using assembler code for microcontrollers – I argued that a compiler would always generate larger object code than my tuned assembler code. But now these microcontrollers are available with 16KB, 32KB and beyond of flash memory! So what is the point of saving a few hundred bytes when you still have over half of the flash space as unused? C code is also much easier to write and debug when compared with assembler code.

The CCS compiler was specifically designed to create optimized code for the PIC family of microcontrollers. As well as all of the standard features that you would expect from a quality C compiler it includes built-in functions to support the on-chip features of a PIC microcontroller. A good example is the **#use RS232** directive; here you specify that you need to use a serial port and you give the compiler details such as baud rate and the IO pins that will be used for TX and RX. If the chosen PIC device has a hardware UART then the compiler will use this for printf and scanf functions, else it will include subroutines to manage the low-level bit manipulations for you. Your main program uses printf statements as before. The CCS compiler also contains built-in functions to drive the on-chip ADC and the real time clock. This way you can focus upon WHAT your program is doing and not the lower level HOW.

Don designed the battery powered data logger shown in Figure 5.6 to demonstrate the capabilities of Vinculum-I. The example program uses a serial connection to control Vinculum-I, which writes

data to the flash drive. Better still, the hardware connection is a standard 4-wire serial port using TX, RX, RTS and CTS.

The PIC runs an application program that has access to a flash drive using Vinculum-I, a real time clock, a temperature and humidity sensor and two analog input channels. A connector for the ubiquitous TTL-232R cable is included, as it a connector for a PIC debugger such as CCS's ICD-U40 unit.

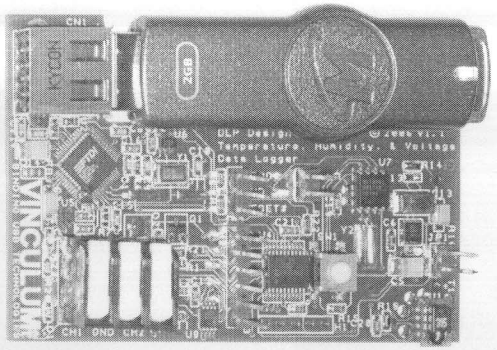


Figure 5.6: The DLP-VLOG showcases Vinculum-I's capabilities

The application program first checks to see if a flash drive is present – if one is not found then the PIC goes back to sleep since there is no point collecting data if there is no where to store it. Once a flash drive is found the PIC starts a data collection cycle: it first reads the real time from the Dallas/Maxim DS1302, then the two analog signals and the battery voltage, then the temperature and humidity. This data is then written to the flash drive and the system goes back to sleep to save battery power. This cycle repeats while a flash drive is present and the battery is charged. The flash drive may be removed at any time and the collected data may then be analyzed using a PC, Mac or Linux system

The source code for the application is available with the Development Kit so that you can customize the data collected and the time interval between samples. Don designed the board as an evaluation tool for Vinculum-I designs but I can see many applications where this battery-operated, portable data logger would be a great fit as is.

Embedded flash drive designs now enabled

I hope that I have shown you that projects built around a flash drive are now easy. Vinculum-I encapsulates all of the required industry standard specifications and presents a simple DOS-like command line interface that is accessed via a serial port (or SPI or parallel FIFO). You add your favorite microcontroller with an application program to control the Vinculum-I peripheral. I presented a few projects to fuel your imagination. My examples used a Cypress PSoC and a MicroChip PIC but the code is readily ported to a different microcontroller architecture. Your project can collect data that is later analyzed on a desktop system or it can be used to redistribute data that was created on a desktop system via lower cost platforms. Project data may be updated by simply swapping flash drives.

If you can read and write to a serial port then, with Vinculum-I, you can read and write data files on flash drives. I would be interested to hear about projects in which you have creatively used a Vinculum and a flash drive.

Chapter 6: Getting to know Vinculum-II

Vinculum-II is FTDI's second generation dual USB host controller and it is a superset of the original Vinculum described in chapter 5 - in fact, the 48 pin LQFP version is backwards compatible (although it needs different firmware). Intense customer feedback on the Vinculum requested more package options, higher performance with lower power and more capability. Vinculum-II delivers on all of these aspects with the added ability to be user programmable - this allows Vinculum-II to support an additional standalone usage model as shown in Figure 6.1. Here the Vinculum-II CPU is also the application CPU and an adjunct microcontroller is not required which will reduce system costs.

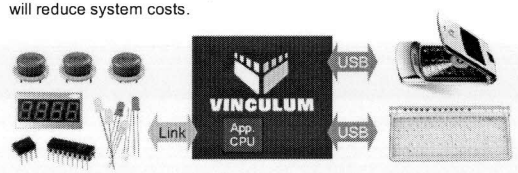


Figure 6.1: Vinculum-II supports standalone operation

Vinculum-II was designed from the ground up to be an efficient C machine and the much larger transistor budget was spent adding hardware assist to all of the peripheral components. Figure 6.2 shows a block diagram of the Vinculum-II, all of function blocks are enhanced over the original Vinculum-I device and there are several new blocks.

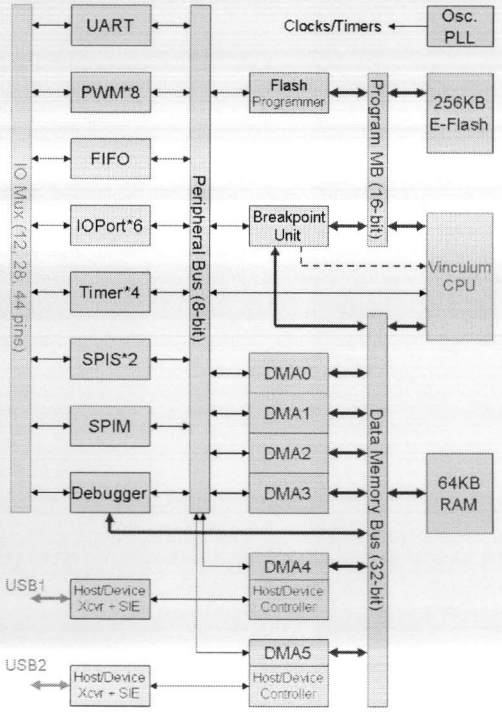


Figure 6.2: Vinculum-II hardware block diagram

The heart of Vinculum-II is a modern 16-bit Harvard Architecture CPU that controls three major buses – a 32-bit data memory accessing 16KB of RAM, a 16-bit program memory accessing 256KB of Flash and an 8-bit peripheral bus. All buses are pipelined and

support concurrent operation. The CPU has no registers (or it has 8K registers depending upon your point of view) and the instruction set has been designed around single clock memory accesses. The instruction set was designed to efficiently implement C code and the user is not expected to use an assembler (although one is available). In fact, all of the examples in part 2 are written in C and I haven't even opened my assembler guide!

The debugger block can take control of the CPU if necessary using the breakpoint unit. There are three hardware breakpoint registers that can trap program or data accesses in real time. The debugger port is a single pin, bi-directional 1Mbaud serial connection and it can take control of the CPU and all internal buses; it can also manage two special peripherals, the breakpoint unit and the flash programmer, enabling a blank Vinculum-II to be easily brought to life with minimal external hardware. Figure 6.3 shows the debugger module that connects to a Vinculum-II target system: the Vinculum-II DIP modules have matching pins and an FTDI Ap Note describes how to integrate this capability into your custom design. This module connects to the development PC using a standard USB cable and this will be demonstrated in Chapters 9 and 10. Note that the Vinculum-II Evaluation Board has the debugger module circuitry built in.

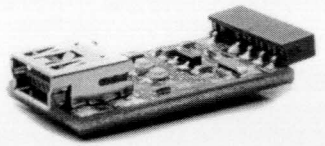


Figure 6.3: A debug module connects to your target system

A major Vinculum-II design goal was efficient power management and, following a reset, only the CPU, clocks, debug port and flash memory are powered. The CPU starts at 48 MHz and can be switched down to 12 MHz or it can move into standby mode where it only consumes about 150 uA. The 12MHz active current is 4.3 mA increasing to 11 mA at 48 MHz.

All the peripherals shown in Figure 6.2 have individual power connections that are not activated unless required by your application program. Each peripheral has one or two control/status registers and these are considered as part of the CPU core - this enables the CPU

to power down a peripheral that is infrequently used while maintaining its state for quick re-activation. Data buffering and movement is handled by a sophisticated DMA controller.

The six - channel DMA controller can move data between memory and peripheral devices or it can implement queues, FIFO or circular, in memory. The CPU is rarely involved in data movement - it sets up DMA channels and responds when data transfers have completed. Four channels are available for application use and two are dedicated to each USB host/slave controller.

The USB host controller is modeled upon the OHCI (Open Host Controller Interface) Specification (a free download from http://www.compaq.com/productinfo/development/openhci.html) with most of the queue handling, error checking and retries implemented in specialized hardware. This results in minimal interaction required by the CPU to support full and low speed data transfers on both host channels simultaneously. Each host controller can also run in slave mode to present a USB device interface to an external host (we shall see examples of both in later chapters).

The Vinculum-II can be used in attached mode (see Figure 5.1) where an external CPU uses the UART, SPI or FIFO peripherals to communicate with a firmware monitor program running on Vinculum-II. FTDI plan to port all of the original Vinculum-I firmware packages (VDAP, VDIF, VCDC, VMSC, and VDPS) into Vinculum-II versions and the Vinculum-II monitor will be the subject of a future FTDI Applications Note.

Additional peripherals and modes have been added to Vinculum-II to support operation in standalone mode (see Figure 6.1). A high-speed synchronous mode has been added to the FIFO function; the original SPI slave modes are supported and a standard, 'unmanaged' slave mode has been added; two slave SPI channels are now available and a master SPI channel has been added; the UART is unchanged.

Vinculum-II also includes: five general purpose 8-bit IO ports where a transition on each bit can also generate an interrupt; 4 general purpose 16-bit timers, each with an optional 16-bit prescalar, that operate in a variety of modes including one shot, continuous and interrupt generation; a 32-bit WatchDog timer that will reset the CPU if not periodically cleared and 8 PWM channels that supports a variety of modes.

Vinculum-II is available in 6 packages types (32/48/64 pin, LQFP/QFN) so, after selecting which peripherals you need for your application, you would choose the smallest (and therefore cheapest) package option for your design. To allow this package flexibility Vinculum-II includes an IO Mux that is used to map peripheral resources onto physical device pins. Full cross-bar switching (i.e. any peripheral pin connectable to any physical pin) consumes a great deal of die area so, in order to keep costs down, the peripheral pins are grouped into sets of four related functions and these functions are connectable to the physical pins in groups of four. Figure 6.4 shows two examples of connecting peripherals to physical pins.

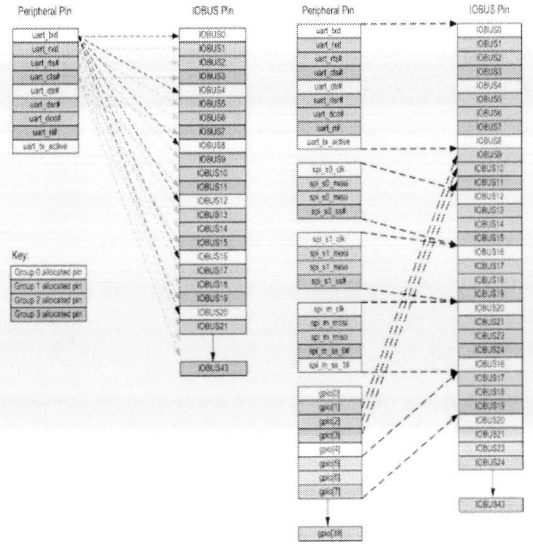


Figure 6.4: The IO Mux connects peripherals to physical pins

A schematic of each physical pin connection is shown in Figure 6.5 and is designed to accommodate a variety of load situations. The output stage operates at 3.3V levels (and is 5.0V tolerant) and may be configured to have a slow or fast (default) slew rate and a drive strength of 4mA (default), 8mA, 12mA or 16mA. An input may be configured as a normal input (default) or a Schmitt trigger input and have no termination (default) or to use a pull up or pull down 75KOhm resistor.

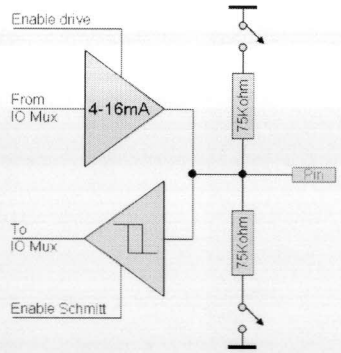


Figure 6.5: Each IO pin has a configurable driver/receiver

Vinculum-II has a fast CPU, ample on-chip program and data memory and a diverse collection of peripherals. For many applications these ample on-chip resources will enable a single-chip Vinculum-II solution. In all cases the peripherals are implemented with a lot of specialized hardware which allows the software to easily control these devices. FTDI also provide device drivers for each of the peripherals and this is the subject of the next chapter; your applications code will not access peripheral registers etc. but will use a common API on top of a micro-kernel. The micro-kernel manages all of the peripherals with tasks that are higher priority than user tasks. In this way, you need not be concerned that your selection of peripherals could change the system timing of, say, the host controller – you can focus upon your applications code.

Chapter 7: Writing software for the Vinculum-II

Writing software for a USB host controller may seem like a daunting task but I should point out that we have already done this! With example 1, in chapter 2, we wrote a program that interacted with the USB host controller on a PC (Windows, OS X or Linux). The physical hardware was masked by the operating system which presented us an API (Application Program Interface). We will follow the same methodology when writing software for our embedded host - FTDI provide an API for Vinculum-II which masks the intricate details of the embedded host controllers, and other hardware resources, so that we can focus on our application program. Figure 7.1 shows the basic structure of the different host environments - note the similarities.

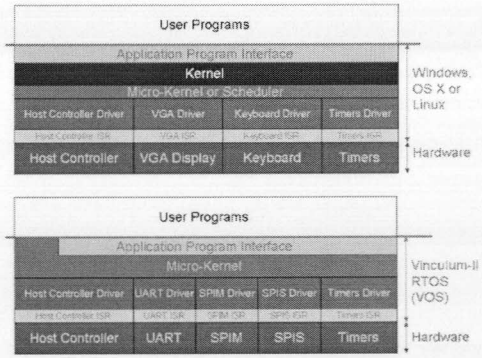


Figure 7.1: Applications programming environments

The lowest levels of drivers within the Windows, OS X, and Linux environments have a micro-kernel or scheduler that must deal with the real time nature of USB host controller communications but this level of detail is not exposed at the API level. Since FTDI expects many Vinculum-II applications to operate with real-time constraints they decided to expose a micro-kernel API to the user. They also provide device drivers for all of the on-chip peripherals. The

combination of micro-kernel plus device drivers is called the Vinculum-II Real Time Operating System, VII-RTOS, or just VOS.

If you haven't written programs using a multitasking RTOS framework before you will discover that it is a good methodology to build applications that do more than one thing, and I can't think of a previous embedded project that I have done that would not have benefited from this approach. If you are familiar with terms such as Task, Thread, and Semaphore you can skip the next section.

Multitasking RTOS 101

You have to learn some new words and concepts to be successful with a multitasking RTOS. This will take some effort so let me first explain the benefit of becoming familiar with these new terms.

You probably write your code using flow charts or state machines. Flow charts are good for describing sequential processes while state machines are good if there are small numbers of possible states with well-defined transition rules. However, both are poor at describing more complex systems with several interdependent parts. Multitasking, on the other hand, is a good fit for such systems - you define a task to handle each part then define how the parts interact.

A significant weakness of the sequential and state machine approaches is that they are inflexible. A good programmer can initially create a workable solution using these approaches but as requirements change and enhancements are demanded the workable design invariably turns into spaghetti code that is difficult to debug and even worse to maintain. The multitasking RTOS approach forces code that is structured so that it can grow and change easily. Changes are implemented by adding, deleting or changing some tasks while leaving other tasks unchanged. Since your code is compartmentalized into tasks, propagation of changes through the code is minimized. This will also reduce testing efforts. So, you have some hard work now to save time and effort later - this is a good deal.

The first paradigm shift you need to make is to partition your program into a set of smaller tasks - each will do one job and it will do it very well. You must also be comfortable with data structures since an RTOS will use a lot of them. Note too that ***task*** now has a specific meaning, it consists of a collection of code bytes that is the program, a collection of variables that are data bytes on the stack and a data structure, also on the stack, called the task **context**. A ***thread*** is a data structure used to describe a task and its operational status.

If you have two, or more, identical peripherals (Vinculum-II has several duplicate units) you can define two threads each with the same code object but with different stack and context objects.

Once your application is divided into several tasks you will define how these tasks interact. The primary inter-task communications mechanism is a ***semaphore***, which is another system-defined data structure. Several operations are defined for a semaphore (object) such as ***Initialize***, ***Signal***, and ***Wait***. A task that creates data will signal when it has data while a task that consumes data will wait until a semaphore is set. Figure 7.2 shows a simple embedded program split into multiple tasks, three in this case.

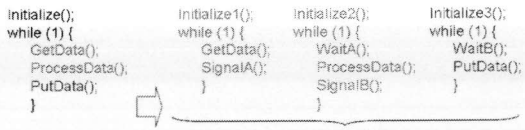


Figure 7.2: A program has several tasks that interact

We will work through an example in the next chapter using real Vinculum-II code rather than the theoretical pseudo-code shown in Figure 7.2 so don't focus upon the details yet. All will become clear with some examples.

Each task is written as if it has sole ownership of the CPU and you must now consider that GetData() runs continuously – mmm, what *did* happen to input data while you were processing and outputting data before? You could now allocate the coding of each task to different programmers with different areas of expertise. Also if a better data processing algorithm is discovered then only one task has to be changed; you need not be concerned about the impacts to the input and output processes since they now operate independent of the processing task. Are you beginning to see some of the benefits of this "divide-and-conquer" approach?

When you divide your program into multiple tasks you will decide that some are more important than others and you can assign these a higher priority. Figure 7.3 shows the classical multitasking RTOS task state diagram – specific details on the Vinculum-II implementation are covered later. As tasks are ***Create***d they are placed on the ***ReadyToRun*** list where the RTOS determines the highest priority task and makes this the ***Running*** task; execution of this task continues until it is blocked for some reason (waiting for a resource, such as a semaphore or a timer) when it is placed on the ***Waiting*** list; the RTOS then places the highest priority task on the *ReadyToRun* list as the *Running* Task; and so the process continues. There is a system-defined task, the ***IdleTask***, which has the lowest priority and is always ready to run.

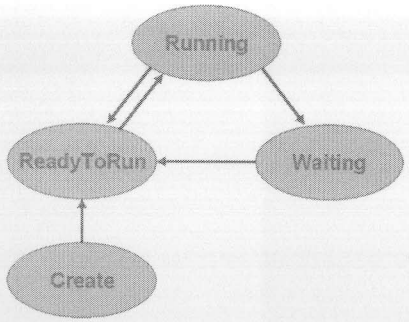


Figure 7.3: Tasks continuously move though this state diagram

Vinculum-II Software Architecture

Figure 7.4 shows a block diagram of the layered Vinculum-II software architecture. This is such an important diagram that I decided to give it a whole page (and I apologize that it is sideways!).

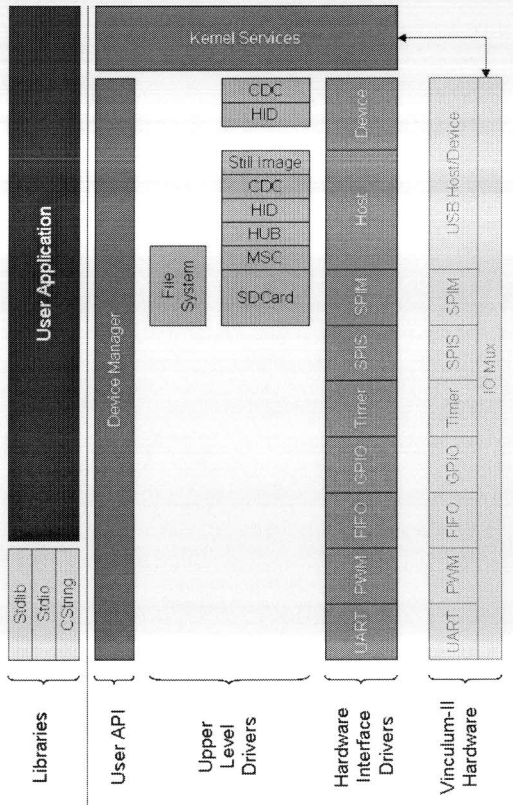


Figure 7.4: Vinculum-II Software Block Diagram

Following a RESET the software environment for Vinculum-II must be set up; the steps taken during this initialization are part of the **kernel services** module of Figure 7.4 and are shown in Figure 7.5. All Vinculum-II programs implement these steps but with different data and, once initialized, the run-time diagram shown in Figure 7.3 describes the operation of your program.

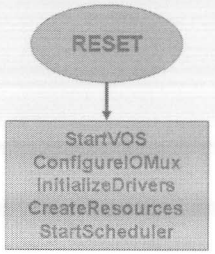


Figure 7.5: Software Initialization Steps

Kernel Services

Looking deeper into the kernel services initialization steps:

StartVOS: this function call initializes all of the internal data structures and sets up the operational parameters of the kernel. The Vinculum-II Operating System, or VOS, needs to know how many device drivers will be used so that it can organize and set aside memory for data buffers. System timing parameters are also set using StartVOS.

ConfigureIOMux: Vinculum-II is available in three package sizes (32, 48 and 64-pin) and this function call sets up the mapping of peripheral Input and Output functions with the physical pins on the package. The number of available IO pins varies with package size (12, 28 and 44) and you cannot get every peripheral signal connected to the outside world on the smallest package. Note too that you should be careful **not** to map away the debugger pin – this is pretty essential for program development and debugging!

InitializeDrivers: at the bottom of Figure 7.4 is the Vinculum-II hardware that was presented in Chapter 6. Each of the peripherals has a set of control and status registers (some more than others) but their intricate hardware details are not exposed since the micro-kernel must own the hardware. Instead, FTDI provides an optimized *Hardware Interface Driver* for each element as shown in the lower level of Figure 7.4. Each driver is tuned for the particular peripheral and handles the device interrupts. A uniform API is presented to the user (described later in this Chapter) to standardize and simplify the application program. Before a driver can be used its driver must be initialized. The driver for most peripherals is small but the USB Host

Controller driver, for example, will set up several threads to manage the root hub, optional downstream hubs and transaction list processing.

CreateResources: a Vinculum-II program will consist of multiple independent tasks that interact with each other. During initialization the threads for each of these tasks will be created along with the semaphores, mutexes and, perhaps, shared buffers needed for inter-task communications. Each thread has a context (object) and handles for shared objects, such as semaphores, may be provided as data within this context. Each thread has its own stack that is initialized with a known pattern so that VOS can track memory usage.

StartScheduler: once all of the program objects have been initialized we start the real time operating system which schedules tasks according to the run-time task state diagram as shown in Figure 7.3. The VOS scheduler uses a round-robin, priority-based, pre-emptive algorithm to run the highest priority task. It also tracks statistics such as thread CPU usage and this enables your application to be profiled and tuned if necessary.

Additional Device Drivers

Returning again to Figure 7.4, notice that layered above the hardware interface drivers are a set of USB Class and Other drivers. FTDI provides (at the time of writing) Mass Storage Class (MSC), HUB, HID, Communications Device Class (CDC) and Still Image drivers on top of the host controller and also HID and FT232 drivers on top of the USB device controller. This means that, out of the box, the Vinculum-II can control flash drives, cell phones, cameras, mice, keyboards, joysticks, etc etc and can also operate as a HID or as an FT232 device. More drivers will be added in future VOS releases.

For advanced users, you can write your own device driver layered on top of the hardware interface driver. FTDI provides an SD Card example layered on top of the SPI-Master driver – this enables immediate support of SD Cards or even the Atmel DataFlash component since this uses the same SPI interface (and is included in the examples in Chapters 9 and 10). Hopefully, by the time that you are reading this, I will have completed an Ethernet driver talking via SPI to the Wiznet W5100 integrated Ethernet Controller!

File System Driver

Layered on top of the MSC driver and the SD Card driver is a FAT file system driver. It supports devices with FAT12, FAT16 or FAT32 structures and include everything you need to simply open,

read or write and then close data files. It handles all of the file allocation tables and directory updates. It only supports 8.3 filenames but I don't see this as an issue for an embedded system. The flash drives, or SD Cards, that Vinculum-II uses are interchangeable with Windows, OS X and Linux systems, as you would expect. I did discover that writing in blocks that are a multiple of the base sector size does give you better performance. The API supports random length file reads and writes but this does cause the driver to run read-modify-write cycles on the physical device. I would recommend doing your own sector buffering as the examples in Chapters 9 and 10 do.

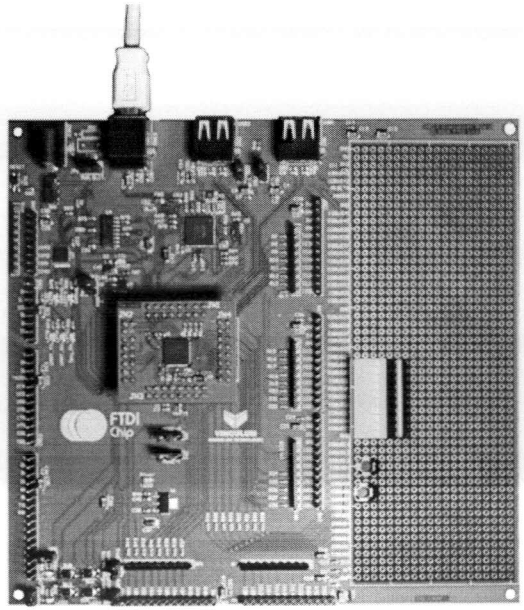
Device Manager

The next level of Vinculum-II software, shown in Figure 7.4, is the Device Manager that provides a consistent and standard interface to the underlying on-chip peripheral device drivers and any added device drivers. The API includes Open, Close, Read, Write and IOCtl (IO Control) functions. All devices are accessed using these standard API functions so communicating over the UART is the same as communicating over SPI, or the USB Host for that matter! This will standardize and simplify your application code and make it easier to change your hardware to match what your marketing team has (over)sold. Any differences between peripherals, such as setting the baud rate of the UART, are handled by the IOCtl API function. The read and write functions are used to stream data to and from devices and four DMA channels are available for user applications. Each host controller also has a DMA channel which the driver uses to move data into and out of any specified user data buffers. These read and write requests can be any length since the file system driver handles all USB packet size issues.

Above the kernel level is another FTDI supplied block; these are standard C run-time support such as string handling, ctype handling, stdlib support and stdio support (fopen, fclose, fread, fwrite that use the FAT file system API described above). The only non-FTDI supplied block is the user application which you write using one or more threads and how to do this is the subject of the next Chapter. FTDI supplies an application program template and several examples to get you up and productive with your Vinculum-II as quickly as possible. I have built on these FTDI examples with two chapters of examples following a Vinculum-II tool chain tutorial which is covered in the next chapter.

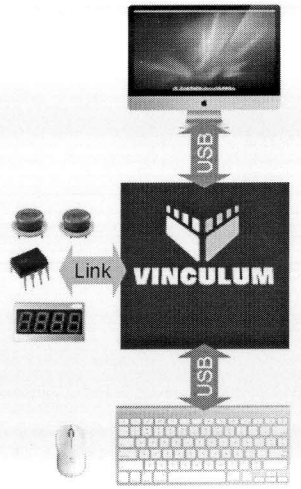
Chapter 8: Using the Vinculum-II IDE (available late May)

This chapter will step through the Vinculum-II IDE using a simple Buttons-And-Lights example – this enables us to focus upon the **tool** and not the application program. This will be an IDE tutorial that shows code creation, generation and debug. The program will be debugged on the V2-Evaluation Board with push-buttons and LEDs added.



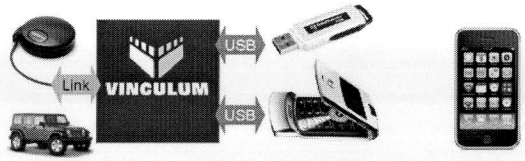
Chapter 9: Building a 'Smart Device' (available late May)

This chapter will step through an example that connects a keyboard to a USB host – this will demonstrate USB Host and USB device drivers. The example will start as a 'key catcher' with keystrokes and mouse movements recorded in an Atmel DataFlash and will later add buttons and lights to create an "Input recorder and Playback unit". The program will be debugged on a V2DIP2-32 module connected to a VNC2 Debug module.



Chapter 10: Interconnecting two USB devices (available late May)

This chapter will step through an example that connects two USB devices together – this will demonstrate support of most of the USB device classes. The example will start as a digital sound recorder that interconnects a microphone with a flash drive. It will also show a Digital Still Camera. The final example connects to a cell-phone and an SMS messaging application is run such that pressing a button will send a text message and receipt of a text message will light an LED. A GPS sensor is also added. The program will be debugged on a V2DIP2-48 module connected to a VNC2 Debug module.



Appendix A: Additional Documentation

For your convenience a selection of FTDI's application notes and datasheets have been included on the CDROM. They are organized:

Documentation/
- Install Guides/
  - Installation on Windows
  - Installation on Mac OS X
- Applications Notes/
  - AN_108 Command Processor for MPSSE Modes
  - AN_109 Programming Guide for High Speed FTCI2C DLL
  - AN_110 Programming Guide for High Speed FTCJTAG DLL
  - AN_113 Interfacing FT2232H Hi-Speed Devices to I2C Bus
  - AN_114 Interfacing FT2232H Hi-Speed Devices to SPI Bus
  - AN_129 Interfacing FT2232H Hi-Speed Devices to JTAG TAP
  - D2XX Programmer's Guide
- Technical Notes/
  - TN_100 USB Vendor ID/Product ID Guidelines.
  - TN_107 FTDI Chipset Feature Comparison.
- Datasheets/
  - FT232R
  - FT245R
  - FT2232H
  - FT232 Cables
  - Vinculum I (VNC1L)
- Utilities/
  - FT_PROG.exe
  - AN_124 User Guide For FTDI FT_Prog Utility

The FTDI EEPROM programming utility, FT_PROG, has also been included on the CDROM. For the most up to date version of this utility, please download it from
http://www.ftdichip.com/Resources/Utilities.htm#FT_Prog

Appendix B: Examples, PCBs and Schematics

The example source code can be found in the Examples/ directory.

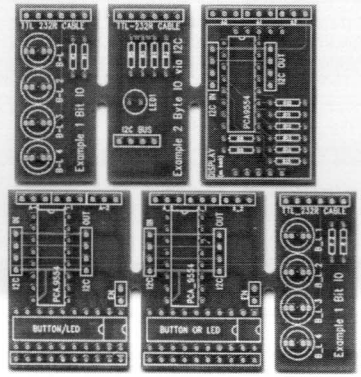
For your convenience datasheets for the third party components used in each of the examples are included in each example sub-directory.

Installing the examples

The examples are provided in multi-platform source code which may be copied and used on your PC. I have provided the Windows versions as Visual Studio project files and the OS X/Linux versions as XCODE project files. The source code is identical for all platforms and the project files will enable you to get up and running instantly.

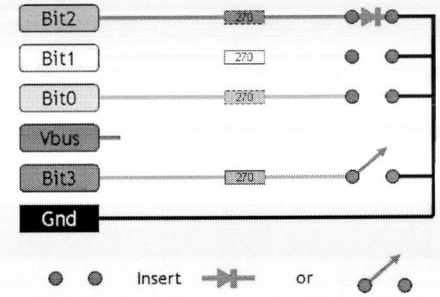
PCBs

A set of simple PCBs are available to try the examples. If you don't have these then the following schematics may be used to construct your own or to create the circuitry on a solder-less breadboard.



Schematics

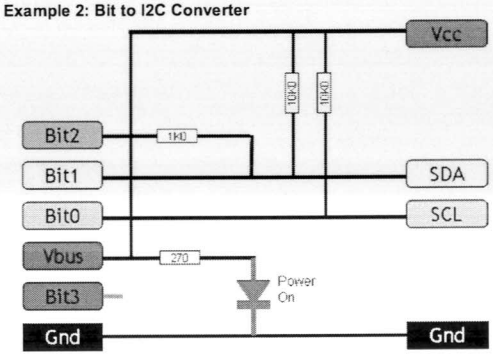
Example 1: Bit IO



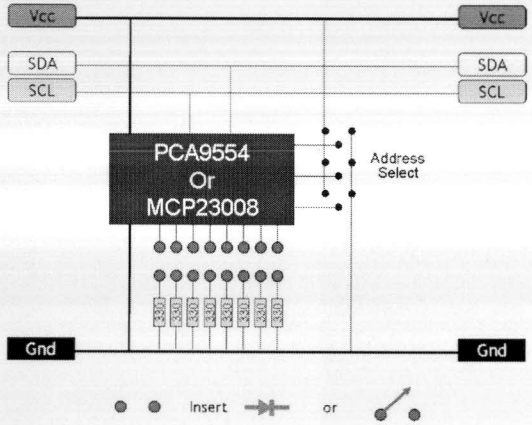
Notes:

Dotted resistors are inside the TTL-232R cable
Note colours of cable connection

Example 2: Bit to I2C Converter



Example 3: I2C to 8 bit port



Notes:

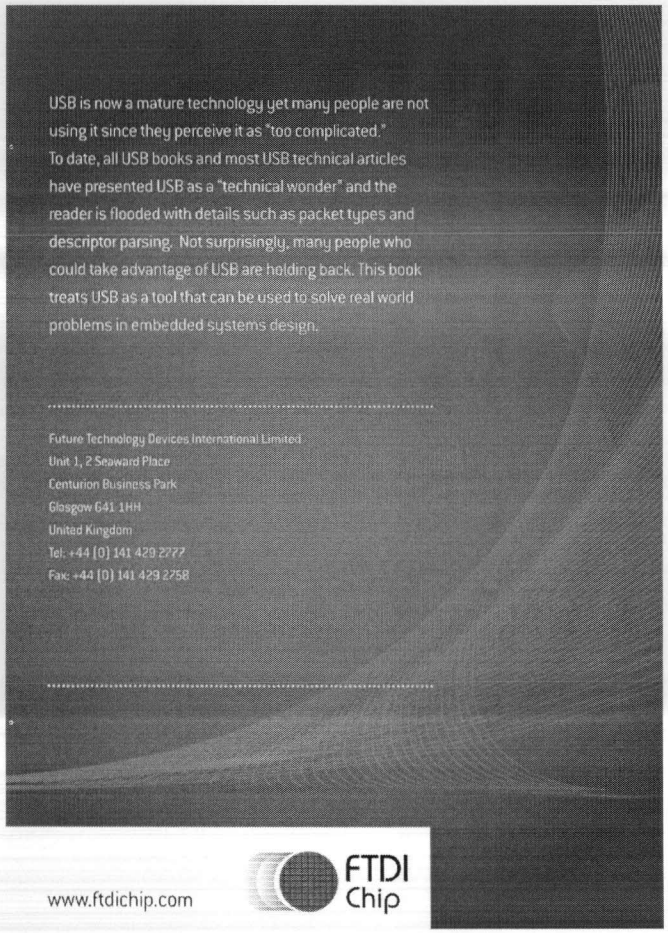
There are two PCBs, one is laid out to accept a DIL button or LED and the other to accept a large 7 segment display (see data sheet in Example3 directory).

PCA9554 and MCP23008 have 3 address pins. By setting these to a binary value between 0 and 7 up to 8 of these devices may be cascaded. Note that you must not exceed 500mA draw from VCC.

Change History

Changes from Revision 1.0

- Various typographical errors fixed
- Chapter 2 examples extended
- Chapter 2 examples implemented on solder-less breadboard
- Chapter 4 example implemented on solder-less breadboard
- Chapters 5, 6 and 7 added


USB is now a mature technology yet many people are not using it since they perceive it as "too complicated."
To date, all USB books and most USB technical articles have presented USB as a "technical wonder" and the reader is flooded with details such as packet types and descriptor parsing. Not surprisingly, many people who could take advantage of USB are holding back. This book treats USB as a tool that can be used to solve real world problems in embedded systems design.
Future Technology Devices International Limited
Unit 1, 2 Seaward Place
Centurion Business Park
Glasgow G41 1HH
United Kingdom
Tel: +44 (0) 141 429 2777
Fax: +44 (0) 141 429 2758
www.ftdichip.com
FTDI
Chip